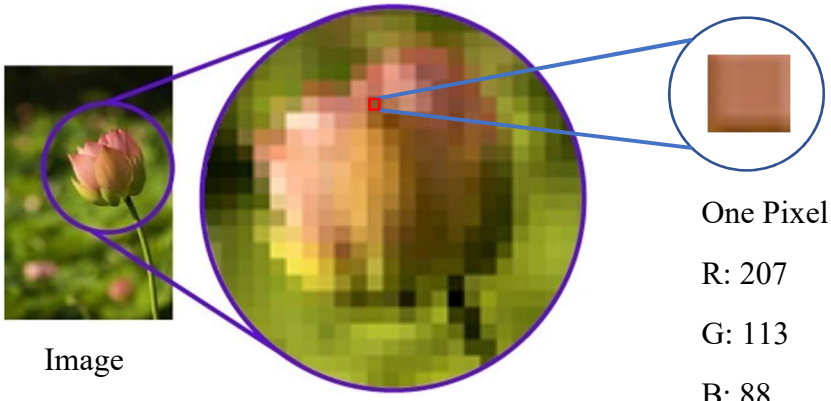
EHB110E Fall 2020-2021 Due Date: 01/18/2020 23:59

**PROGRAMMING ASSIGNMENT – 2**

In this programming assignment, you will implement basic image processing in C.

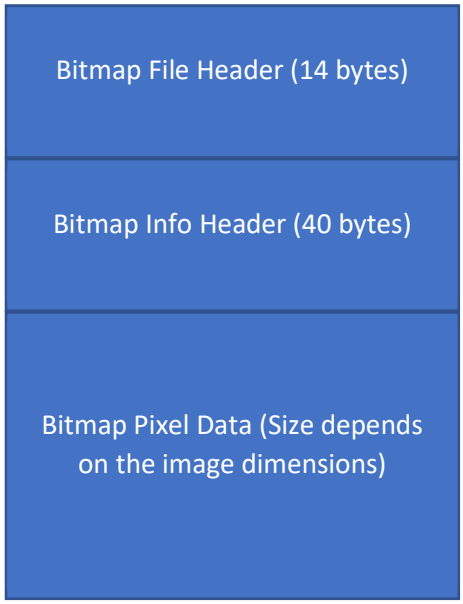
Below picture represents an image structure.




An image is made up of a two-dimensional grid of individual pixels as shown above. Each pixel has a color represented as a combination of red, green and blue color components (RGB). Each color component has an intensity value of 1-byte which translates to a range between 0-255. An intensity of zero means no intensity and an intensity of 255 means full intensity. As the intensity of a color component increases, its contribution to the resulting color increases in parallel. As an example, a full red color has R, G, B values of 255, 0, 0, respectively. A full black color has R, G, B values of 0, 0, 0 each. A full white color has R, G, B values of 255, 255, 255. By combining various intensity values of R, G, B color components, one can obtain pixels having different colors which when combined as shown above forms up a full image.

Images can be stored in different formats on a computer such as bitmap (BMP), JPEG, PNG, etc. In this assignment, we will work with BMP images, specifically 24-bit uncompressed BMP files. By 24-bits, it is meant that each pixel is 24-bits wide. That is, 8-bits for each color component of a pixel.

The structure of a BMP file is shown below:



The Bitmap file header has information about file type, file size, etc. The Bitmap Info header, on the other hand, has information about the image such as image dimensions, pixel width, etc. The detailed structure of these headers are shown below and are shared with you in a bitmap.h file in this project. For the purpose of this project, we will only be interested in the **bfSize**, **biWidth** and **biHeight** fields of which descriptions are given below.

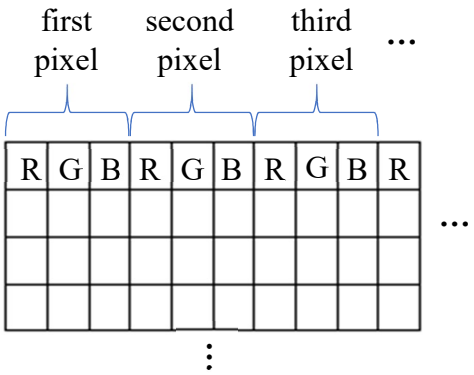
```
typedef struct {
  uint16_t  bfType;
  uint32_t bfSize;
  uint16_t  bfReserved1;
  uint16_t bfReserved2;
  uint32_t bfOffBits;
} BITMAPFILEHEADER;
```

**Relevant Members**
**bfSize:** The size, in bytes, of the bitmap file.

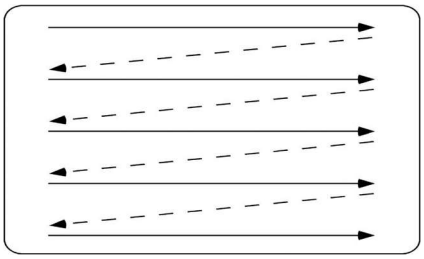
```
typedef struct {
  uint32_t biSize;
  int32_t  biWidth;
  int32_t  biHeight;
  uint16_t  biPlanes;
  uint16_t biBitCount;
  uint32_t biCompression;
  uint32_t biSizeImage;
  int32_t    biXPelsPerMeter;
  int32_t    biYPelsPerMeter;
  uint32_t biClrUsed;
  uint32_t biClrImportant;
} BITMAPINFOHEADER;
```

**Relevant Members**
**biWidth**: Specifies the width of the bitmap, in pixels.
**biHeight**: Specifies the height of the bitmap, in pixels.

After the headers, comes the pixel data. This is raw pixel bytes (3-bytes per pixel) in raster scan order. That is, the first line of the image starting from the left-most pixel comes first, followed by the pixels on the right on the same line, followed by the left-most pixel of the second line of the image, followed by the remaining pixels on the right on the second line and so on (see figure below).



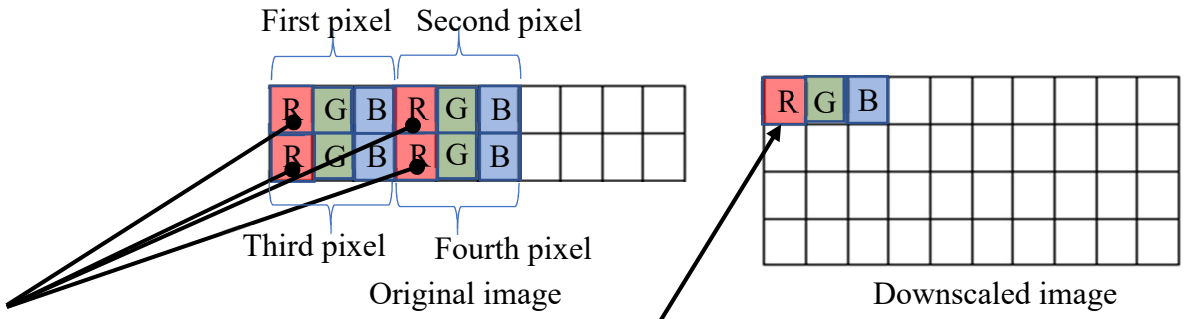

**Pixel data of an image**



**Order of pixels in the bitmap pixel data section of the BMP file**

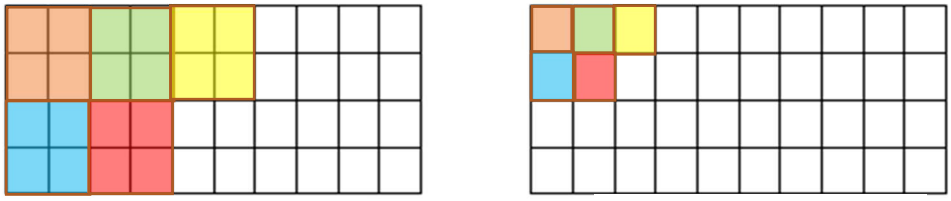
In this first part of the programming assignment, you will read a BMP image from a file provided to you (itu.bmp), and you will scale the image down by a factor of 2x in each dimension (the image size will be halved both horizontally and vertically). In order to scale the image, you are supposed to implement a basic downscaling algorithm described below:

You will take the average of the color component values of neighboring 4 pixels and that average will be the resulting color component of the pixel in the downscaled image. You will repeat this for each color component and for each 4 pixel group. The below figure shows the process.




This figure shows individual bytes of each pixel. The red color bytes of these 4 neighboring pixels are averaged to produce one red color byte written here.

Same is done for green and blue color bytes resulting in 1 RGB pixel to be produced out of 4 RGB pixels

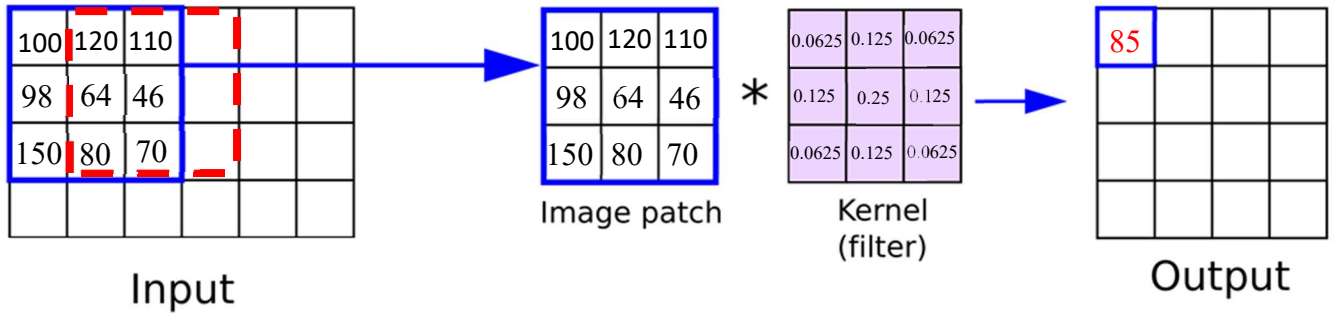



This figure shows pixel view of the same operation (each grid square corresponds to 1 pixel). 4 neighboring pixels are component-wise averaged together to produce 1 pixel in the downscaled image on the right-hand side.

Finally, you will write the resulting pixel values to an output BMP file. You need to write the bitmap file header as well as bitmap info header before the pixel values. Note that you need to modify the header fields mentioned above to account for the reduced image size.

**PROGRAMMING ASSIGNMENT – 3**

In this programming assignment, you will apply a smoothing filter on the original image. In order to apply smoothing, you need to apply a convolution operation on the image. A convolution operation uses a kernel which is a fixed size small matrix. For this assignment, we will use a 3x3 blurring kernel. The convolution operation moves this kernel over the image, shifting it **one pixel** at a time and takes the dot product of matrix elements with the pixel values underneath (element-wise multiplication and addition operation – see below). The filter traverses the entire image this way producing a single pixel out of 9 adjacent pixels.

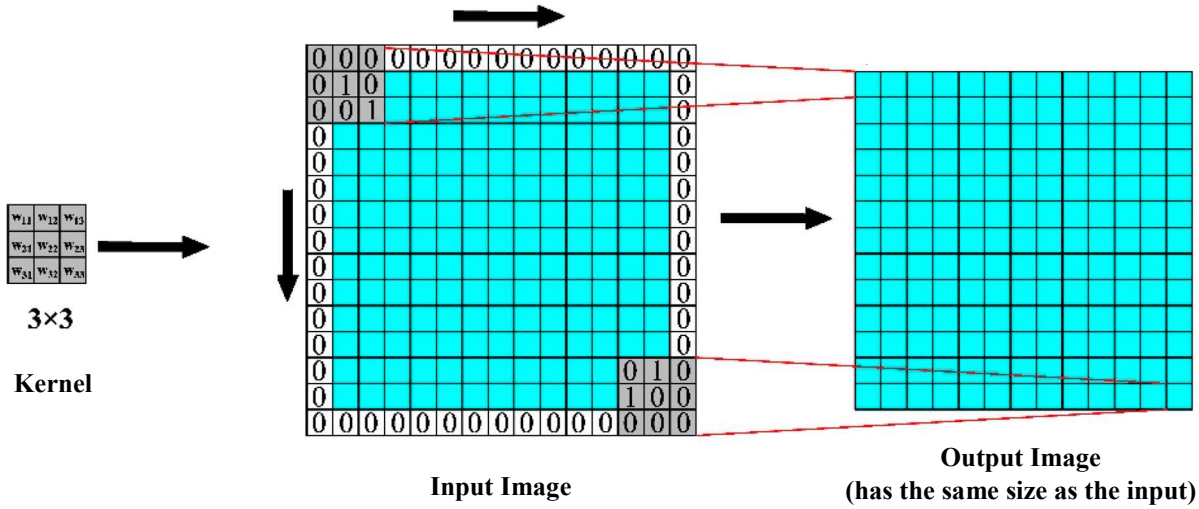



$100\times0.0625+120\times0.125+110\times0.0625+98\times0.125+64\times0.25+46\times0.125+150\times0.0625+80\times0.125+70\times0.0625 = 85.875 \sim 85$

**In this figure, each grid box represents a single pixel for simplicity.**

Note that similar to downscaling operation, this operation needs to be carried out separately for each color component of a pixel. That is, firstly, the kernel will be applied to R color components of 9 adjacent pixels producing the R color component of the resulting pixel; then, it will be applied to G color components of the same 9 adjacent pixels producing the G color component of the resulting pixel; and finally, it will be applied to B color components of the same 9 adjacent pixels producing the B color component of the resulting pixel.

However, there is a problem. If we do not let the kernel move beyond the boundaries of the image, the resulting image will be smaller than the original image. For instance, a 3x3 kernel traversing a 90-pixel wide image left-to-right can only produce 90 - 2 convolutions without kernel exceeding image boundaries. Therefore, in this case, the resulting image would have a width of 88 pixels. In order to preserve the image size both horizontally and vertically, we apply padding to the input image by adding 0 valued pixels around the edges of the image. See below.



**Deliverables and Requirements**

1. Source code for Programming Assignment-2 and Programming Assignment-3 in a single C file (you will submit a single C file that implements both programming assignments).
2. Source code must be commented properly.
3. Your code needs to read the input image file and should produce two output image files in BMP format one for the downscaled image (Programming Assignment-2) and another one for the smoothed image (Programming Assignment-3). You can name these output files as itu-downscaled.bmp and itu-smoothed.bmp
4. Your code needs to use dynamic memory allocation to allocate memory for processing images in memory and should free that memory properly when done.
5. Use functions to modularize your code.
6. Do not include binaries in submission.