This homework has a required part and an optional part. The required part is 100 points. If you do the optional part you will get upto 40 points extra (In other words if the optional part is correctly done you get 40 points extra).

1 Digital Images

A digital image is simply a rectangular grid (array) of squares. Each square is of a single solid color and is known as a pixel (short for picture element). The reason why we do not see individual pixels as squares is because they are very small. Resolution of a display device is measured by pixel density, or number of pixels per inch (PPI). Digital displays marketed from 2009 onwards have at least 100 PPI, with the newer and sharper ones reaching 200300 PPI. Pixels are commonly arranged in a two-dimensional grid, the dimensions of which are specified by the images size in pixel resolution. For example, when we have a JPEG image of 1920x1080, it means the image has 1920 pixels in its width and 1080 pixels in its height. It has 1920x1080 = 2,073,600 total pixels and thus can also be referred to as a 2-megapixel image.

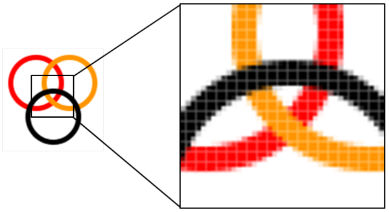

Figure 1: An image showing individual pixels rendered as squares

The color representation of a pixel typically requires more than a single bit (which can only represent two colors, black and white), and the number of bits used per pixel is known as color depth. Colors are encoded as integers from 0 to 255, where 0 is black (no color) and 255 is full color and use the RGB color model in which red, green and blue are (added) blended together to produce a broad spectrum of colors. Consult the wikipedia page `https://en.0wikipedia.org/wiki/RGB_color_model` For more details on RGB colors. Figure 2 shows three pixels with their individual RGB color values and indices at which they are stored in the pixel grid. For example, the leftmost pixel can be found at row 81, column 123 and is colored with a red value of 137, green value of 196, and blue value of 138. Note that an RGB value of (137, 137, and 138) would be light gray, and thus the combination of (137, 196, 138) comes out light green.

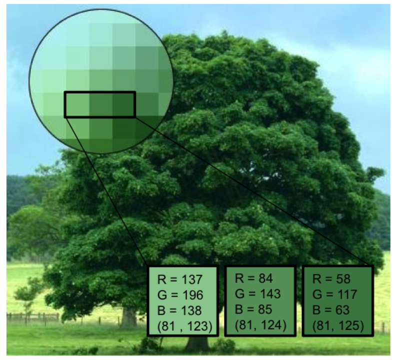

Figure 2: Three pixels

2 Quadtree

A quadtree is a tree data structure in which each internal node has exactly four children and is most often used to partition a two-dimensional space by recursively subdividing it into four quadrants. On an image, a quadtree recursively divides the image into four subimages and stops when some

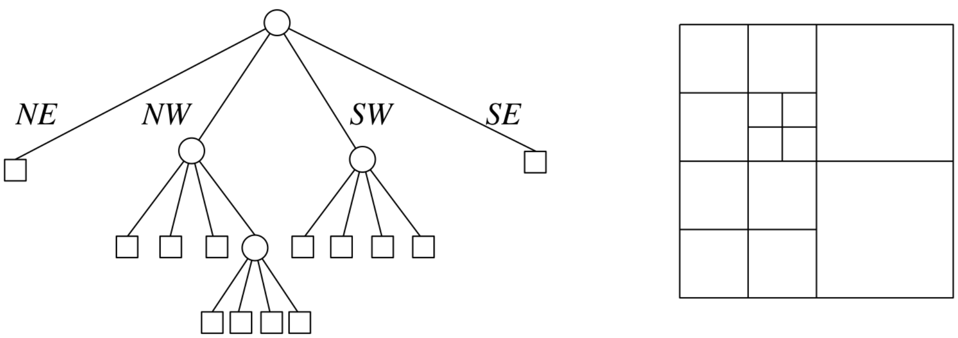

Figure 3: Quadtree

criteria are met or reaching a single pixel. Each node either stores the color of the pixel (if it's a single pixel), or the average of the colors of all pixels in the quadrant.

2.1 Quadtree Image Compression

A quadtree decomposition can be used to compress an image in stages. See the animation at `https://en.0wikipedia.org/wiki/Quadtree#/media/File:Quadtree_compression_of_an_image.gif` which shows the quadtree compression of an image step by step. Also see the sequence of images shown in Figure 4. This computes a hierachy of images that represent a high quality image with increasingly more details (in the right places). The idea is to split only in those quadrants where the colors of the children differ greatly (according to some threshold) from that of the parent. Note the quadtree is selective and regions we choose to not split on presumably already have more or less the same colors and can afford to be blurred into the mean color. When the subdivion stops, the compressed image can be recovered from the leaves of the quadtree. Different settings of the difference threshold will generate compressions at different resolutions.

Besides compression, there are many applications for image hierachies - a form of level-of-detail techniques in Computer Graphics. In computer games for example, if a player is far away, only low-res textures are applied on objects and as the player gets closer, textures of higher and higher resolutions are swapped in.

The algorithm is: split the image into four subimages, for each current node n_i, calculate the mean color C_i and mean squared error $E_i = \frac{1}{N^2}\Sigma_{x=1}^{N}\Sigma_{y=1}^{N}|n_i(x,y) - C_i|^2$. The error E_i is the average of the cumulative squared error between the compressed representation C_i and the original, which is computed as the squared Euclidean distance between the mean color C_i and all original pixel colors in the quadrant n_i of size $(N \times N)$. That is, for a pixel with color (r, g, b), the squared error should be computed as $((r - C_i.r)^2 + (g - C_i.g)^2 + (b - C_i.b)^2$. The usual square root is not needed. If a subimage has error greater than some **threshold**, split into four further subimages.

3 Image Processing

This section is for the OPTIONAL part. Now that you see an image as an array of colors (given as integer triples), image processing is as simple as looping over this array and changing the numbers! This is known as "filtering". Some simple image filters are for example:

1. negative - change each pixel `(r, g, b)` to `(255-r, 255-g, 255-b)`
2. grayscale - change each pixel `(r, g, b)` to `(c, c, c)` where `c = r*0.3+g*0.59+b*0.11`
3. tint - given a tint color `(R, G, B)`, scale each pixel color `(r, g, b)` to `(r/255*R, g/255*G, b/255*B)`. Note that full white `(255, 255, 255)` will become exactly `(R, G, B)`, and everything else will be scaled proportionally between 0 and R/G/B.

3.1 Convolution Filters

A another popular technique is to compute a pixel's color based on its immediate neighbors, including itself. We will base our discussions of these filters on a 3x3 neighborhood, the smallest and simplest; however, in practice neighborhoods can be larger, as well as differently shaped than a square (box). A 3x3 box filter computes the pixel color via a weighted average of all 9 pixels using some predetermined weights. If we consider an input pixel $input[i, j]$, then the weighted average can be defined as:

$$\begin{aligned} output[i,j] = {} & w_1 \times input[i-1, j-1] + w_2 \times input[i, j-1] + w_3 \times input[i+1, j-1] \\ & + w_4 \times input[i-1, j] + w_5 \times input[i, j] + w_6 \times input[i+1, j] \\ & + w_7 \times input[i-1, j+1] + w_8 \times input[i, j+1] + w_9 \times input[i+1, j+1] \end{aligned}$$

The choice of the weights (kernel) has dramatic effect on the resulting image. One popular and useful kernel is edge detection, with the weights set as:

$$\begin{matrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{matrix}$$

That is:

$$\begin{aligned} output[i,j] = & -input[i-1,j-1] - input[i,j-1] - input[i+1,j-1] \\ & - input[i-1,j] + 8 \times input[i,j] - input[i+1,j] \\ & - input[i-1,j+1] - input[i,j+1] - input[i+1,j+1] \end{aligned}$$

Edge detection works by enhancing the differences between the center pixel and those that surround it. When you perceive an edge in an image, you are simply noticing that there is a (sharper) change in color/brightness. Thus, edge detection works by setting a pixel to black (0) if its not very different from its neighbors, and trend towards white (255) the more different it is. Consider what happens when the filter above is applied to an area of similar colors. The sum will trend towards zero. If the pixel is high contrast and bright, the sum will become more positive.

4 Requirements

1. Given an input image (We provide kira.ppm as a test image), create a quadtree decomposition of the image and compute 8 compressed images of increasing resolutions as explained in 2.1. These 8 images should be generated at compression levels of approxiamately 0.002, 0.004, 0.01, 0.033, 0.077, 0.2, 0.5, 0.65. The compression level is defined as the number of leaves in the quadtree divided by the number of the original pixel count. Note lower levels represent higher compression ratio. **The computed ratios do not have to equal the numbers exactly, approximately is fine. You will experimentally find the threshold (to stop splitting) to approximately achieve the required compression level.That is, you keep track of the number of leaves/number of pixels ratio at all times, and determine whether you need to lower or raise the current threshold. Depending on the image, it is possible that some compression levels are not possible, for example those that have many similar colors and lack contrast. In those cases, simply output a message to console and only generate the images corresponding to those levels that are achievable.**

2. It is acceptable to only work on square images. If the input image is not square, print appropriate error message and exit.

3. **OPTIONAL FOR EXTRA CREDIT:** Edge detecion is expensive, since the filters require multiple (9 for 3x3, 25 for 5x5, etc) operations per pixel. With high resolution digital images having pixel counts in the 10s of millions and results needed in real time, it is often preferred to only apply the filter in important areas, which is where the quadtree comes in. Based on the same quadtree decomposition of an image, apply the edge detection filtering as explained in 3.1 only to those nodes of a sufficiently small size and replace the color of the larger nodes with black.

5 Images Formats

In this assignment, our input will be images given in the Portable Pixel Map (PPM) format, which is a simple text file listing colors of each pixel in an image, as explained here: `https://en.0wikipedia.org/wiki/Netpbm_format` Your output will be image(s) in the same format. Note that there are two versions of PPM, P3 (also known as plain or ascii) and P6 (binary). P6 is much more widely found because it is more space efficient and less prone to parsing difficulties. On the other hand P3 is plain text and thus easier to debug. We will be using the P3 format. Most image readers will convert to/from ppm to any other standard image format. On Linux, the command is called `convert`, which is part of the `imagemagick` software suite. An image called `test.png` can be converted using the following command: `convert test.png -compress none test.ppm`. You can substitute `png` for any other populuar image format extensions such as `jpg` and `tif`. For more information, consult the manual pages using `man convert`.

6 Command Line Input

You will receive an image file on the command line following the `-i` flag as your input: `java Main -i test.ppm`. In addition, support the following flags:

1. `-o <filename>` indicates the root name of the output file that your program should write to
2. `-c` indicates that you should perform image compression
3. **OPTIONAL:** `-e` for edge detection

For example:
`java Main -c -i test.ppm -o out` will generate 8 compressed images of `test.ppm` named `out-1.ppm`, `out-2.ppm`, ..., `out-8.ppm`, where `out-1.ppm` is the image with the lowest resolution/highest compression and so on. `java Main -e -i test.ppm -o out` will generate one output image called `out.ppm` which is the result of applying edge detection to `test.ppm`. You may assume that only one of `-c` or `-e` will be given. Order of the flags should not matter, i.e. `java Main -o out -e -i test.ppm` is equivalent to `java Main -e -i test.ppm -o out`

7 Where to Start

At this point, you should be able to design a class structure on your own, so I do not give you specific suggestions here. However, here are some hints about tasks that you should make sure you can perform:

1. Be able to read in an image from a given file name and write out an image to a given file name.
2. Take an image and generate a quadtree (both compressed and not compressed).
3. **OPTIONAL** Apply filters to an image.

Important notes regarding submission:

- To submit your homework, store everything in a directory whose name is your name. In particular, DO NOT send directories with names like homework4. Then zip the directory and email us a single attachment.
- In addition to your source files, include in your submission directory a README file: There must be a file called README, which contains any helpful information including:
  - Your name.
  - How to compile and run your program:
  - Known Bugs and Limitations: List any known bugs, deficiencies, or limitations with respect to the project specifications.
  - File directory: If you have multiple source or data files, please explain the purpose of each.
- Please delete all .class files prior to submission.

START RIGHT AWAY!

Figure 4: Quadtree compression of an image step by step (8 steps, row by row from left to right). Left image shows the compressed image with the tree bounding boxes while the right shows just the compressed image

Figure 5: edge detection