# What to Look For in Problem Set 4: Filter (More/Less)

## What to Look For

### Evidence of a 5-point Submission

- **Minimal iterations through the image**; at most once for **grayscale** and **sepia**, at most twice for **blur** and **edges**, and at most one-half (up to **width / 2**) for **reflect**
- **Minimal casting, rounding, square rooting**; no more casts or rounds than are strictly necessary. You should only need a single **(float)** to do the math needed in this assignment
- **Clean handling of edge cases**; **blur** handles its edge cases by changing the bounds for the innermost two for loops, NOT by only running the contents of those two loops based on a condition (note that edges does have to use an inner condition)
- **Ideal edges implementation**; uses a 2D array to store the kernel, uses the **&&** operator to avoid extra conditional statements, efficient iteration in inner loops

### Evidence of a 4-point Submission

- like a 5-point submission, but with some small mistakes, such as:
    - a few extraneous roundings, castings, or other arithmetical operations
    - might use three lines to copy a pixel from one place to another instead of one; i.e., might do
        - `image[i][j].rgbtRed = new_image[i][j].rgbtRed`
        - `image[i][j].rgbtRed = new_image[i][j].rgbtRed`
        - `image[i][j].rgbtRed = new_image[i][j].rgbtRed`

      instead of `image[i][j] = new_image[i][j]`
    - might use a condition instead of different loop bounds to execute the inner two for loops in blur
    - might not use a helper function to handle certain things, akin to the **max**/**min** helper functions in the staff solution

### Evidence of a 3-point Submission

- like a 4-point submission, but with one or two more serious errors, such as:
    - duplicates the image where it isn't necessary; i.e., makes a new array in any function except for **blur** and/or **edges**
    - extraneous iteration; **grayscale** or **sepia** iterate more than once, or **blur** or **edges** iterate more than twice, or **reflect** iterates further than **width / 2**
    - excess conditionals that make the code much harder to read

## Evidence of a 2-point Submission

- Makes a copy of the input image in any function *except* **blur** and **edges**
- **grayscale** function may unnecessarily use three separate variables to store a given pixel's red, green, and blue values, may use unnecessary conditionals or computation to calculate the average
    - Unnecessary parentheses
    - More than one **round**
    - More than one cast
- (less) **sepia** function may use a **for** loop to iterate over the 3 RGB values, may use unnecessary conditionals or computation to calculate the sepia values
    - Unnecessary rounding
    - Uses unnecessary conditionals that add extra computation
- **reflect** function copies RGB values instead of the entire pixels, uses a **while** loop to iterate over half of the image
- **blur** function iterates over the height and width three or more times, may include many conditions to check for edge cases
- (more) **edges** function iterates the height and width three or more times, may include unnecessary conditional statements

## Evidence of a 1-point Submission

- **grayscale** function may iterate over the height and width more than once, may not use the **round** function (manually rounding)
- (less) **sepia** function may iterate over the height and width more than once
- **reflect** function iterates over more than half of the image (more than width / 2)
- **blur** function hard-codes edge cases
- (more) **edges** function hard-codes many cases, repeats large portions of code (instead of grouping cases together)

# Example Implementations (Worse vs. Better)

## **grayscale** Function

### Worse Implementation

The example below uses three unnecessary variables for RGB values, casts more than once and includes an unnecessary conditional when calculating the average

```
for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        int red = image[i][j].rgbtRed;
        int green = image[i][j].rgbtGreen;
        int blue = image[i][j].rgbtBlue;
        float avg = (((float) red + (float) green + (float) blue) / 3);

        int max_color_value = 255;
        int average = 0;
        if (avg < max_color_value)
        {
            average = round(avg);
        }
        else
        {
            average = max_color_value;
        }

        image[i][j].rgbtRed = average;
        image[i][j].rgbtGreen = average;
        image[i][j].rgbtBlue = average;
    }
}
```

### Better Implementation

The example below uses one variable to reference the given pixel and calculates the average in a neat and efficient way (no extraneous rounding or casting)

```
for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        RGBTRIPLE pixel = image[i][j];
        int avg = round((float) (pixel.rgbtRed + pixel.rgbtGreen + pixel.rgbtBlue) / 3);

        image[i][j].rgbtRed = avg;
        image[i][j].rgbtGreen = avg;
        image[i][j].rgbtBlue = avg;
    }
}
```

## **sepia** Function (less)

### Worse Implementation

The lengthy example below unnecessarily rounds three times for each sepia value, inconsistently initializes some variables outside of the **for** loop and some variables inside the **for**

loop. May also be improved by eliminating the **else** statements (instead, just using the **if** statements to reassign the sepiaColor value)

```
int sepiaGreen;
int sepiaBlue;
int sepiaRed;
for (int i = 0 ; i < height; i += 1)
{
    for (int j = 0; j < width; j += 1)
    {
        int green = image[i][j].rgbtGreen;
        int blue = image[i][j].rgbtBlue;
        int red = image[i][j].rgbtRed;

        sepiaRed = round(.393 * red) + round(.769 * green) + round(.189 * blue);
        if (sepiaRed > 255)
        {
            image[i][j].rgbtRed = 255;
        }
        else
        {
            image[i][j].rgbtRed = sepiaRed;
        }

        sepiaGreen = round(.349 * red) + round(.686 * green) + round(.168 * blue);
        if (sepiaGreen > 255)
        {
            image[i][j].rgbtGreen = 255;
        }
        else
        {
            image[i][j].rgbtGreen = sepiaGreen;
        }

        sepiaBlue = round(.272 * red) + round(.534 * green) + round(.131 * blue);
        if (sepiaBlue > 255)
        {
            image[i][j].rgbtBlue = 255;
        }
        else
        {
            image[i][j].rgbtBlue = sepiaBlue;
        }
    }
}
```

### Better Implementation

The example below uses three variables for each RGB value, a **min** helper function, and rounds in each computation just once

```
for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        int red = image[i][j].rgbtRed;
        int green = image[i][j].rgbtGreen;
        int blue = image[i][j].rgbtBlue;

        image[i][j].rgbtRed = min(255, round(red * .393 + green * .769 + blue * .189));
        image[i][j].rgbtGreen = min(255, round(red * .349 + green * .686 + blue * .168));
        image[i][j].rgbtBlue = min(255, round(red * .272 + green * .534 + blue * .131));
    }
}
```

## **reflect** Function

### Worse Implementation

The example below uses a **while** loop instead of a **for** loop (does not iterate over half the width)

```
int left = 0;
int right = width - 1;

RGBTRIPLE temp;
for (int i = 0; i < height; i++)
{
    while (left < right)
    {
        temp = image[i][left];
        image[i][left] = image[i][right];
        image[i][right] = temp;
        left++;
        right--;
    }
    left = 0;
    right = width - 1;
}
```

The example copies over each RGB value individually (instead of the whole pixel). Also includes unnecessary parentheses and an unnecessary return statement

```
for (int i = 0; i < width / 2; i ++)
{
    for (int j = 0; j < height; j++)
    {
        int buffer_red = image[j][i].rgbtRed;
        int buffer_green = image[j][i].rgbtGreen;
        int buffer_blue = image[j][i].rgbtBlue;

        image[j][i].rgbtRed = image[j][(width - 1) - i].rgbtRed;
        image[j][i].rgbtGreen = image[j][(width - 1) - i].rgbtGreen;
        image[j][i].rgbtBlue = image[j][(width - 1) - i].rgbtBlue;

        image[j][(width - 1) - i].rgbtRed = buffer_red;
        image[j][(width - 1) - i].rgbtGreen = buffer_green;
        image[j][(width - 1) - i].rgbtBlue = buffer_blue;
    }
}
return;
```

### Better Implementation

The example

```
for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width / 2; j++)
    {
        RGBTRIPLE tmp = image[i][j];
        image[i][j] = image[i][width - 1 - j];
        image[i][width - 1 - j] = tmp;
    }
}
```

## **blur** Function

### Worse Implementation

This example iterates over the image an unnecessary number of times (inner-two-most **for** loops unnecessarily iterate over the entire image again). Additionally, the counter variable is a float (an int makes more sense here) and unnecessary variables are declared when calculating the average

```
for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        int red = 0;
        int green = 0;
        int blue = 0;
        float counter = 0.0;
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                if (abs(j - x) <= 1 && abs(i - y) <= 1)
                {
                    red = red + copy[y][x].rgbtRed;
                    green = green + copy[y][x].rgbtGreen;
                    blue = blue + copy[y][x].rgbtBlue;
                    counter = counter + 1;
                }
            }
        }

        int red_average = round(red / counter);
        int green_average = round(green / counter);
        int blue_average = round(blue / counter);

        image[i][j].rgbtRed = red_average;
        image[i][j].rgbtGreen= green_average;
        image[i][j].rgbtBlue = blue_average;
    }
}
```

### Better Implementation

This example handles both edge pixels and inner pixels in the same, robust way. Also begins and ends indexing in the **for** loop at the proper pixel indices using **min** and **max** helper functions (no extra calculation needed)

```
for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        int red = 0, green = 0, blue = 0, count = 0;

        for (int row = max(0, i - 1); row <= min(height - 1, i + 1); row++)
        {
            for (int col = max(0, j - 1); col <= min(width - 1, j + 1); col++)
            {
                count++;

                red += image[row][col].rgbtRed;
                green += image[row][col].rgbtGreen;
                blue += image[row][col].rgbtBlue;
            }
        }

        new_image[i][j].rgbtRed = round((float) red / count);
        new_image[i][j].rgbtGreen = round((float) green / count);
        new_image[i][j].rgbtBlue = round((float) blue / count);
    }
}
```

## **edges** Function (more)

### Worse Implementation

The example below includes unnecessary variables and an **if - else** statement that could be avoided with a better choice of starting and ending indexing variables and conditional sub-statements

```
int gx[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
int gy[3][3] = {{ -1, -2, -1}, {0, 0, 0}, {1, 2, 1}};

for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        int redSumX = 0;
        int greenSumX = 0;
        int blueSumX = 0;
        int redSumY = 0;
        int greenSumY = 0;
        int blueSumY = 0;
        int a = 0;
        int b = 0;
        for (int m = i - 1; m < i + 2; m++)
        {
            for (int n = j - 1; n < j + 2; n++)
            {
                if (m < 0 || n < 0 || m >= height || n >= width)
                {
                    b++;
                    if (b > 2)
                    {
                        b = 0;
                        a++;
                    }
                }
                else
                {
                    redSumX += (imagecopy[m][n].rgbtRed * gx[a][b]);
                    greenSumX += (imagecopy[m][n].rgbtGreen * gx[a][b]);
                    blueSumX += (imagecopy[m][n].rgbtBlue * gx[a][b]);
                    redSumY += (imagecopy[m][n].rgbtRed * gy[a][b]);
                    greenSumY += (imagecopy[m][n].rgbtGreen * gy[a][b]);
                    blueSumY += (imagecopy[m][n].rgbtBlue * gy[a][b]);
                    b++;
                    if (b > 2)
                    {
                        b = 0;
                        a++;
                    }
                }
            }
        }
        // the rest of the function beneath here...
```

### Better Implementation

The example below uses a 2D matrix to store the multiplier values, and includes an efficient choice of **for** loop indices and conditional sub-statements to avoid repeating unnecessary code

```
int xkernel[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
int ykernel[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};

for (int i = 0; i < height; i++)
{
    for (int j = 0; j < width; j++)
    {
        int xred = 0, yred = 0, xgreen = 0, ygreen = 0, xblue = 0, yblue = 0;

        for (int x = -1; x <= 1; x++)
        {
            for (int y = -1; y <= 1; y++)
            {
                int xadj = xkernel[x + 1][y + 1];
                int yadj = ykernel[x + 1][y + 1];

                if (i + x < height && i + x >= 0 && j + y < width && j + y >= 0)
                {
                    RGBTRIPLE pixel = image[i + x][j + y];

                    xred += xadj * pixel.rgbtRed;
                    xgreen += xadj * pixel.rgbtGreen;
                    xblue += xadj * pixel.rgbtBlue;
                    yred += yadj * pixel.rgbtRed;
                    ygreen += yadj * pixel.rgbtGreen;
                    yblue += yadj * pixel.rgbtBlue;
                }
            }
        }
        // the rest of the function beneath here...
```