Chapter 5 Homework

This assignment is based on a project created by the textbook authors to go with chapter 5. Their version of the instructions, which includes diagrams and explanations that might be helpful to you, can be found here.

Part 1

Start with the code below, with comments indicating where you need to complete several functions (more practice doing tricky things in C!) before moving on to part 2.

```
/*
 * pixels.c
 *
 * David Owen
 *
 * Based on idea from Bryant and O'Hallaron's performance lab
 * project for chapter 5 of Computer Systems: A Programmer's
 * Perspective.
 */

#define N 5 // (1 << 13) // Use a small value to check
                         // correctness, a big value to check
                         // performance.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef unsigned short pixel;

void pack_pixel(float r, float g, float b, pixel *p)
{
    *p = (((int) (r * 31) & 0x1F) << 11) +
         (((int) (g * 63) & 0x3F) << 5) +
          ((int) (b * 31) & 0x1F);
}

// Complete this function, using pack_pixel to help you
// understand what it should do.  (Notice that the green value
// gets more bits than red or blue.  Read more about this here:
// http://en.wikipedia.org/wiki/High_color#16-bit_high_color
void unpack_pixel(pixel p, float *r, float *g, float *b)
{
}

void pixel_to_gray(pixel p, pixel *q)
{
    float r, g, b, avg;
    unpack_pixel(p, &r, &g, &b);
    avg = (r + g + b) / 3;
    pack_pixel(avg, avg, avg, q);
}

void array_to_gray(pixel *p, pixel *q)
{
    int i, j;
```

```c
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            pixel_to_gray(p[i * N + j], &q[i * N + j]);
}

// Using pixel_to_gray as a model, and the factors listed
// below, complete this function to get a sepia tone version of
// a pixel.  (If the new value for any color exceeds 1, set it
// to 1).
//     new_r = r * .393 + g * .769 + b * .189
//     new_g = r * .349 + g * .686 + b * .168
//     new_b = r * .272 + g * .534 + b * .131
void pixel_to_sepia(pixel p, pixel *q)
{
}

// ...And, once you have a single-pixel sepia function, use it
// to make a sepia tone version of an array of pixels.
void array_to_sepia(pixel *p, pixel *q)
{
}

void transpose(pixel *p, pixel *q)
{
    int i, j;

    for (i = 0; i < N; i++)
        for (j = 0; j <= i; j++)
        {
            q[i * N + j] = p[j * N + i];

            if (j != i)
                q[j * N + i] = p[i * N + j];
        }
}

void flip_vertical(pixel *p, pixel *q)
{
    int i, j;

    for (i = 0; i <= N / 2; i++)
        for (j = 0; j < N; j++)
        {
            q[(N - i - 1) * N + j] = p[i * N + j];

            if (i < N / 2)
                q[i * N + j] = p[(N - i - 1) * N + j];
        }
}

// Using flip_vertical as a guide, complete this function so
// that it creates a horizontally flipped version of an array
// of pixels.
void flip_horizontal(pixel *p, pixel *q)
{
}
```

```
// In part 2 you'll be creating new (and hopefully faster)
// versions of this function.
void rotate1(pixel *p, pixel *q)
{
    pixel *t = (pixel *) calloc(N * N, sizeof(pixel));
    transpose(p, t);
    flip_vertical(t, q);
    free(t);
}

// Complete this function based on the description given below.

// Returns pixel with R, G, B values equal to the average of pixel
// (i, j) and its 8 neighbors' R, G, B values.
//
// For pixel (i, j), these are the pixels that should be averaged:
//
// (i - 1, j - 1) | (i - 1, j) | (i - 1, j + 1)
// ---------------+------------+---------------
// (i, j - 1)     | (i, j)     | (i, j + 1)
// ---------------+------------+---------------
// (i + 1, j - 1) | (i + 1, j) | (i + 1, j + 1)
//
// For cases where (i, j) is too close to the top or bottom, or
// to a side, the pixels that would be beyond the edge should be
// left out.
void average(pixel *p, int i, int j, pixel *q)
{
}

// In part 2 you'll be creating new (hopefully faster) versions
// of this function.
void smooth1(pixel *p, pixel *q)
{
    int i, j;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            average(p, i, j, &q[i * N + j]);
}

// Use this for testing (with small values of N).
void print_array(pixel *p)
{
    int i, j;
    float r, g, b;

    if (N > 6)
    {
        printf("TOO BIG TO PRINT!\n");
        return;
    }

    printf("\n");
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            unpack_pixel(*(p + i * N + j), &r, &g, &b);

            if (r == 0 && g == 0 && b == 0)
                printf("-----------  ");
            else
                printf("%.1f,%.1f,%.1f  ", r, g, b);
        }

        printf("\n");
    }
}

int main(int argc, char **argv)
{
    long start;
    pixel *p = (pixel *) calloc(N * N, sizeof(pixel));
    pixel *q = (pixel *) calloc(N * N, sizeof(pixel));

    // calloc will fill the newly allocated memory with zeros.
    // These statements put some values in around the top left
    // and bottom right corners, just to help with testing for
    // correctness.
    pack_pixel(0.1, 0.1, 0.1, p);
    pack_pixel(0.2, 0.2, 0.2, &p[1]);
    pack_pixel(0.3, 0.3, 0.3, &p[2]);
    pack_pixel(0.4, 0.4, 0.4, &p[N]);
    pack_pixel(0.9, 0.9, 0.9, &p[N * N - 1]);

    // Follow the pattern here to time function calls in part 2.
    start = clock();
    rotate1(p, q);
    printf("rotate1: %.3f seconds\n",
            (clock() - start) / (double) CLOCKS_PER_SEC);

    // Comment these out when you move from checking correctness
    // for small arrays to checking performance for big arrays.
    print_array(p);
    print_array(q);

    return 0;
}
```

Part 2

Using the techniques summarized in section 5.13, and explained throughout the chapter, create new optimized versions of the rotate and smooth functions. Create multiple numbered versions of each, adding comments to explain what optimizations you tried, so that you get credit for your work even if some of the things you try don't improve performance.

You may experiment with different levels of gcc optimizations (default, `-O1`, `-O2`, `-O3`)—some of the modifications you make may be significant at one level of optimization but not at another.

To give you something to shoot for, see if you can create new versions of `rotate` and `smooth` that run at least twice as fast as the original versions, for at least one of the optimization levels.

Grading

/ 15	pixels.c submitted, shows reasonable effort to complete and test Part 1 modifications.
/ 15	…reasonable effort to complete Part 2: various optimization techniques attempted, code in `main` to compare new versions of `rotate` and `smooth` with the old ones.
/ 20	pixels.c compiles and runs without errors.
/ 20	New version of `rotate` significantly faster than original (at least twice as fast for full credit here).
/ 20	New version of `smooth` significantly faster than original (at least twice as fast for full credit here).
/ 10	pixels.c has comment with student name, and clear and consistent formatting; does not have additional errors missed in test runs.
/ 10	Up to 10 points extra credit for performance improvement beyond what's required (i.e., better than twice as fast). Overall grade for assignment won't exceed 100.