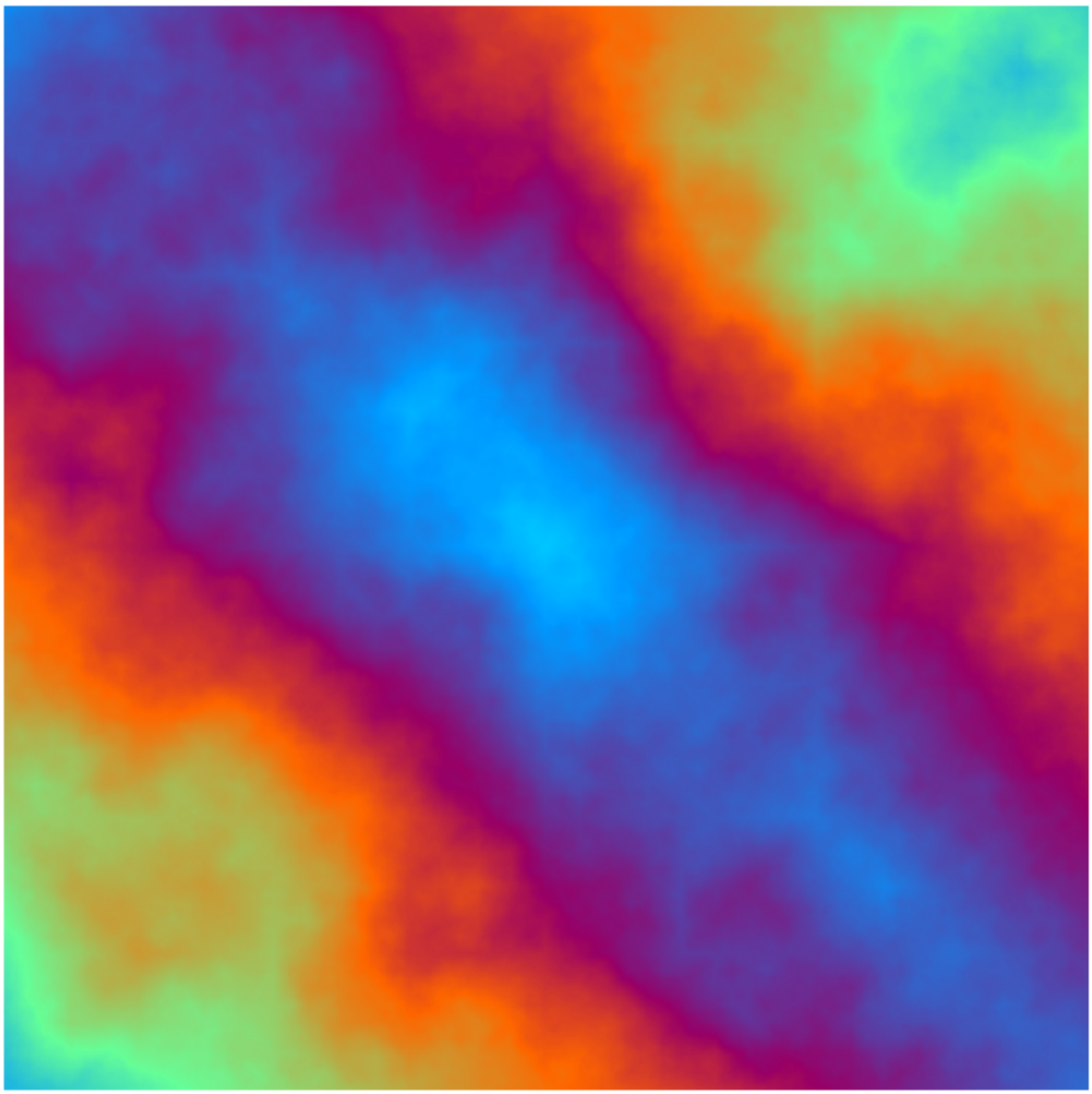
Note that your **`drawMandelbrotSet`** function itself is not directly recursive. You can use loops to iterate over the the range of pixels of interest in the window. But the code that examines each pixel to determine whether or not it diverges must be recursive for full credit and should not use loops. You should probably put such code into a helper function that is called by **`drawMandelbrotSet`**.

Debugging: It can be hard to debug this problem because if your calculations are not quite right, you may see no graphical output at all. If you are not seeing any helpful output, try inserting a **`cout`** statement for each pixel to verify whether you have properly figured out its corresponding complex number. Also try printing out the number of iterations until each complex number diverges, to help you see whether you are computing that value properly.

4. Plasma Fractals

A Plasma fractal, also known as random midpoint displacement, is a recursive algorithm used to generate superbly cool-looking psychadelic gradients. Take a look:

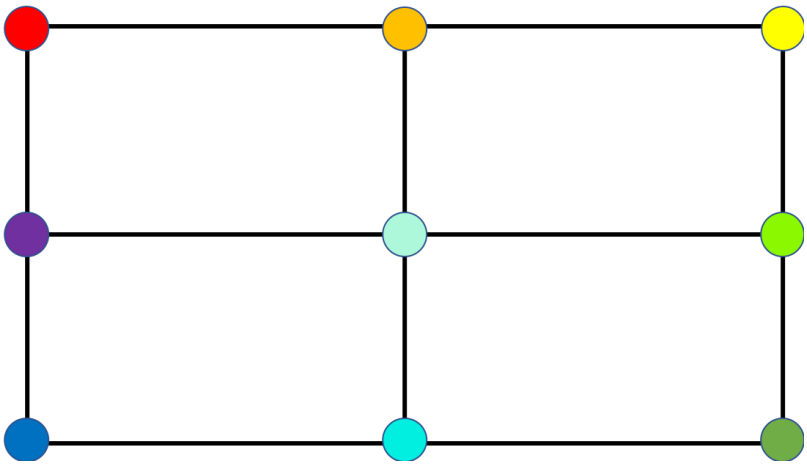


The algorithm works at a high level by recursively sub-dividing the rectangular area you wish to fill into 4 equal-sized sub-rectangles until you get down to about 1 pixel-sized rectangles, at which point you color each pixel following certain coloring rules. The coloring rules are such that colors change randomly across the fractal, but gradually, to create the appearance of a smooth gradient. The algorithm behind this is used in real-world scenarios to generate terrain maps, which could be used in areas like video game development or movie production (e.g. imagine that lighter colors are higher, and lower colors are lower terrain).

Here's how the algorithm works. Initially, take the rectangle you wish to fill and assign 4 random colors to it, one to each corner. You can visualize this like the following:

If the rectangle has a width or height of at most 1, then it's small enough we can just fill it in. You calculate its color by averaging the colors of its 4 corners. If the rectangle is too big, we need to subdivide it first. So we divide this rectangle into 4 smaller rectangles:

Now we need to process each of these subrectangles. But before we do that, we need to know what each of their corner colors are. We can do this by, on each edge where we need a new color, averaging the colors of the two endpoints of that edge, like so:



The middle point is the lone exception. This point is the average of all 4 colors in the corners, and ***randomly displaced*** by a certain amount. This means we add a small random number to it that will slightly change the color. This provides the subtle color gradient you see in the fractal.

Now we continue by processing each of the 4 sub-rectangles we just divided, with their corresponding corner colors. Once we divide all rectangles down to a small enough size, we fill them in and we get a pretty cool-looking fractal.

The user will select an area in the provided GUI that they wish to fill, and the information will be passed to your function.

To help with the implementation, we have provided a variable type called **`PlasmaColor`** that makes handling these colors easier. Here is its functionality.

Method	Description
`PlasmaColor()`	Constructor that creates a new **`PlasmaColor`** with a random color.
***`color`*`.toRGBColor()`**	returns the color this variable represents as an int that can be put into a pixel grid.
`color1 + color2`	returns a color that is the sum of 2 colors.
`color1 / decimal`	returns a color that is the color divided by the provided decimal.
`ostream << color1`	Outputs the color as a decimal between 0 and 1 to an output stream.

For example, you can create a new random **`PlasmaColor`** as follows:

```
PlasmaColor myColor;
```

If you'd then like to get half of that color and get the corresponding int for that color, you could do the following:

```
myColor /= 2.0;
int colorInt = myColor.toRGBColor();
```

See the included **`plasmacolor.h`** file for a full overview of functionality.

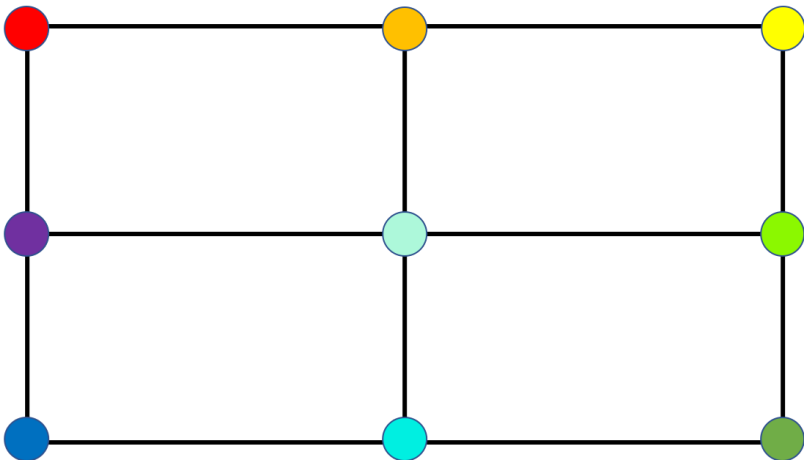
To calculate the middle color of a rectangle at a given step, use the provided **`displace`** function in the starter code:

```
displace(double newWidth, double newHeight, double totalWidth, double totalHeight)
```

The parameters are:

1. **`newWidth`**: the width of the subdivided rectangles you are creating at that moment
2. **`newHeight`**: the height of the subdivided rectangles you are creating at that moment
3. **`totalWidth`**: the width of the **entire** plasma fractal (not just the part you are filling in at the moment)
4. **`totalHeight`**: the height of the **entire** plasma fractal (not just the part you are filling in at the moment)

As an example, let's say I am filling in a plasma fractal of total size 80 x 40 and am processing the following rectangle, where the width is 20 and the height is 10.



At this step, I subdivide into 4 rectangles of size 10 x 5. To calculate the middle color, I take the average of my 4 corners, and then **add** the displacement value calculated by the above function to that to get the final middle color. For that specific call, the parameters would be **`displace(10, 5, 80, 40)`**.

You may ask why this displacement function takes these parameters. The displacement function gives you a random value that varies the color slightly. For larger rectangles (e.g. ones that are closer to the size of the total fractal), it varies the color more than for smaller rectangles (e.g. ones that are closer to 1), where it displaces very little. This causes the effect of significant color changes over the entire fractal, but only small changes in local areas of the fractal.

You may not use **any loops or data structures** for this problem other than the already-provided pixel grid. You must use recursion to fill in each pixel in the fractal.

Because this fractal is generated with randomness, it's not possible to provide expected output. However, your fractal should have a smooth color gradient across the image, and look mostly random (and not like a pattern). Here are some more

examples below. One tip for debugging is to print out colors to ensure your calculations seem correct. You can output a color to an output stream, where it will display as a decimal between 0 and 1. This is how they are represented under the hood, so you can verify all math by ensuring that the resulting numbers come out correctly (e.g. averaging colors is just averaging their decimal values).