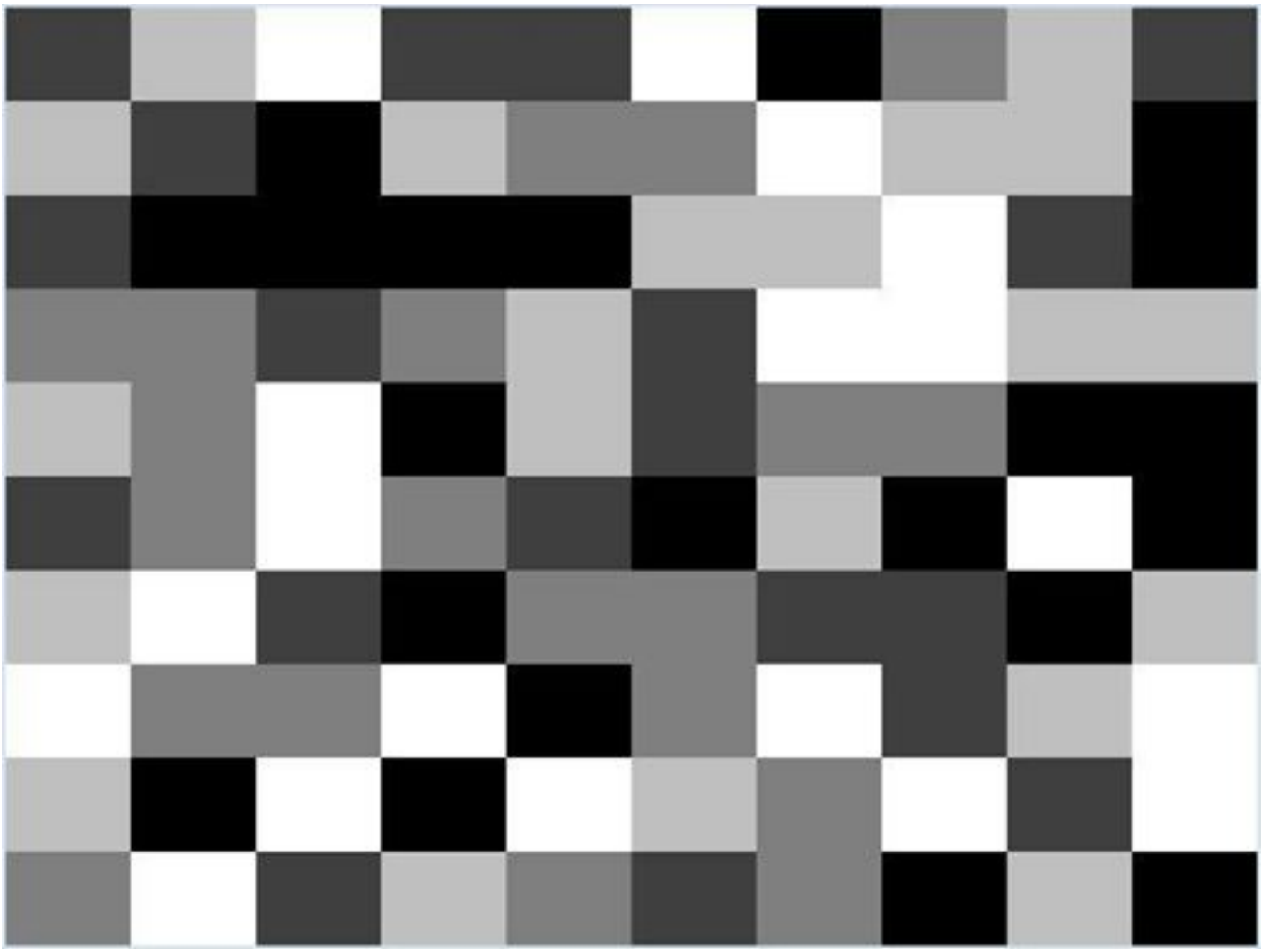

Background Theory

Ramsey theory, named after the British mathematician and philosopher Frank P. Ramsey, is a branch of mathematics that studies the conditions under which order must appear, often out of chaos. Ramsey theory questions are typically of the form: "how many elements of some structure must there be to guarantee that a particular property will hold?" Ramsey theory is the study of the preservation of properties under set partitions. In other words, given a particular set S that has a property P, is it true that whenever S is partitioned into finitely many subsets, one of the subsets must also have property P?

Much of the exposition that follows is from *Ramsey Theory on the Integers*, 2nd ed., by Landman and Robertson.

Ramsey theory is a further refinement of the Generalized Pigeonhole Principle: If more than mr elements are partitioned into r sets, then some set contains more than m elements.

Proof. Let S be a set with $|S| = mr$. Let $S = S_1 \cup S_2 \cup \cdots \cup S_r$ be any partition of S. Assume, for a contradiction, that $|S_i| \leq m$ for all $i = 1, 2, \ldots, r$. Then $|S| = \sum_{i=1}^{r} |S_i| \leq mr$, a contradiction. Hence, for at least one i, the set S_i contains more than m elements, i.e., $|S_i| \geq m + 1$.

Here is an example: For each integer $n = 1, 2, \ldots, 200$, let $R(n)$ be the remainder when n is divided by 7. Then some value of $R(n)$ must occur at least 29 times. To see this, we can think of the 200 integers as the pigeons, and the seven possible values of $R(n)$ as the pigeon-holes. Then, according to above theorem, since $200 > 28(7)$, one of the pigeonholes must contain more than 28 elements.

Here's an example which is a little more obscure: Color each point in the xy-plane having integer coordinates either red or blue. We show that there must be a rectangle with all of its vertices the same color. Consider the lines $y = 0, y = 1$, and $y = 2$ and their intersections with the lines $x = i, i = 1, 2, \ldots, 9$. On each line $x = i$ there are three intersection points colored either red or blue. Since there are only $2^3 = 8$ different ways to color three points either red or blue, by the pigeonhole principle two of the vertical lines, say $x = j$ and $x = k \neq j$ must have the identical coloring (i.e., the color of (j, y) is the same as the color of (k, y) for $y = 0, 1, 2$). Using the pigeonhole principle again, we see that two of the points $(j, 0), (j, 1)$, and $(j, 2)$ must be the same color, say (j, y_1) and (j, y_2). Then the rectangle with vertices (j, y_1), (j, y_2), (k, y_1), and (k, y_2) is the desired rectangle.

In the last example, we used colors as the "pigeonholes." Using colors to represent the subsets of a partition in this way is often convenient, and is quite typical in many areas of Ramsey theory, though we could just as well have used integers.

Here's another example. We will prove the following: at a party of six people, there must exist either three people who have all met one another or three people who are mutual strangers (i.e., no two of whom have met). By the pigeonhole principle, we are guaranteed that for each person, there are three people that person has met or three people that person has never met. We now want to show that there are three people with a certain relationship between them; namely, three people who all have met one another, or three people who are mutual strangers. First, assign to each pair of people one of the colors red or blue, with a red "line" connecting two people who have met, and a blue "line" connecting two people who are strangers. Hence, we want to show that for any coloring of the lines between people using the colors red and blue, there is either a red triangle or a blue triangle (with the people as vertices). To this end, single out one person at the party, say person X. Since there are five other people at the party, by the pigeonhole principle X either knows at least three people, or is a stranger to at least three people. We may assume, without loss of generality, that X knows at least 3 people at the party. Call these people A, B, and C. So far we know that the lines connecting X to each of A, B, and C are red. If there exists a red line between any of A, B, and C then we are done, since, for example, a red line between A and B would give the red triangle ABX. If the lines connecting A, B, and C are all blue, then ABC is a blue triangle.

In this project we ask the question, in an n by n array of k different characters, can we find a rectangular sub-array whose corners are all the same characters? Must such a sub-array exist, or is there an array that contains no such sub-array? For example, a 7 by 7 array of three characters may not contain such a sub-array, There are $3^{49} \approx 2.4e23$ such arrays to search through, and the search that produced the following example involved about a million instances before finding the one shown to the right:

```
3 1 2 1 2 3 2
3 1 3 3 3 1 2
2 3 3 1 2 3 1
1 3 1 1 3 2 2
1 1 2 2 3 3 1
2 3 2 3 1 2 1
2 1 1 2 1 3 2
```

To make things more interesting, we'll use the SFML library to display a checkerboard of colors, instead of numbers. Your program should provide the following:

1. Two fields for text entry on the graphics page where the user can enter values for n and k;
2. Mouse clickable "start" and "stop" buttons on the graphics page to initiate or interrupt the search for an $n \times n$ array of k colors that does not contain a sub-matrix whose corners are all the same color.
3. A display of the $n \times n$ matrix of colors as the search commences.
4. A counter indicating how many matrices have been searched so far without success: "success" meaning that you have an $n \times n$ array with no submatrix containing identical corners.
5. A search algorithm (this can be as simple random guess and check.)

Here's some starter code, starting with the Ramsey class

```
// Ramsey.h

#include <cstdlib>
#include <ctime>
#include <iostream>
#include <SFML/Graphics.hpp>

class Ramsey : public sf::Drawable {
    int n; /// dimension of the matrix
    int k; /// number of different characters
    int** M; /// The n by n matrix of 1,2,..,k values
public:
    bool is_valid{1}; //Is the matrix in a valid state?
    bool adjust_n(int offset) {
        // If n+offset < 2 return false: we've under-adjusted.
        // Set is_valid to true since we want to start in a valid state.
        // Delete M (this need to be done for all the subarrays first, and lastly for M itself.)
        // Make the adjustment in n (either up one or down one, depending on offset.)
    // Allocate enough memory to hold your new array.
    // Initialize all elements of M to 1 (it'll be randomized somewhere else.)
    // If we've reduced n to the point where k > n, then set k down to n.
    }
    bool adjust_k(int offset) {
        // If k+offset < 2 or > n, return false (you don't want that!)
        // Otherwise, set is_valid to true.
    // Make the adjustment in k (either up one or down one, depending on offset.)
    // Initialize all elements of M to 1 (it'll be randomize later.)
    }
    Ramsey() {}; //default constructor
    bool testRamsey(int* M[]) {
        for(int i=0; i < n-1; ++i) // for each row
            for(int j = 0; j < n-1; ++j) //for each column
                for(int k = i+1; k<n; ++k ) //for each subsequent row
                    for(int l = j+1; l<n; ++l)   //each subs. column {
                        /*std::cout << "\nM[" << i << "][" << j << "]=" << M[i][j]
                                  << "\nM[" << i << "][" << l << "]=" << M[i][l]
                                  << "\nM[" << k << "][" << j << "]=" << M[k][j]
                                  << "\nM[" << k << "][" << l << "]=" << M[k][l]
                                  << std::endl;*/
                        if(M[i][j]==M[i][l] &&
                                M[i][l]==M[k][j] &&
                                M[k][j]==M[k][l])
                            return false;
                    }
        return true; // it passes the Ramsey test.
    }
    Ramsey(int en, int kay) : n(en), k(kay)  { //constructor
    // Allocate heap memory for an n by n matrix, M, of int.
        // Initialized all elements of the M to 1.
    }
    void display() { // For debugging
        for(int i = 0; i < n; ++i) {
            for(int j = 0; j < n; ++j)
                std::cout << M[i][j] << "  ";
            std::cout << '\n';
        }
    }
    void draw(sf::RenderTarget& target, sf::RenderStates states) const; // draw prototype
    void update()     {
        if (is_valid) return; // If it's good, no need to update
        // Otherwise, set all values of M to random numbers between 1 and k, inclusive.
        if(testRamsey(M))   // If we have the Ramsey state,
          // then set is_valid is to true
    }
```

```
};
void Ramsey::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    for(int i = 0; i < n; ++i) {
        for(int j = 0; j < n; ++j) {
            sf::RectangleShape rect(sf::Vector2f(600/n,600/n));
            rect.setPosition(sf::Vector2f(i*600./n,j*600./n));
            rect.setFillColor(sf::Color(255/M[i][j],255/M[i][j],255/M[i][j]));
            target.draw(rect);
        }
    }
}
```

Then the Game class:

```
#include "Ramsey.h"
#include <vector>

class Game : public sf::Drawable {
private:
    int n; //matrix size;
    int k; //number of characters
    Ramsey* R; //Acces to a Ramsey object
    //Here's a way to handle the buttons that's pretty slick!
    std::vector<std::pair<sf::RectangleShape, void (Game::*)(sf::Event)>> buttons;
    void start(sf::Event e); // The function that gets called when the button is pushed
    void stop(sf::Event e);
    void inc_n(sf::Event e);
    void dec_n(sf::Event e);
    void inc_k(sf::Event e);
    void dec_k(sf::Event e);
public:
    Game(int N = 8, int K = 8) : n(N), k(K) {  //Game constructor
      //Define the buttons' rectangles.
        sf::RectangleShape start_button_rect        (sf::Vector2f(150,60));
        sf::RectangleShape stop_button_rect         (sf::Vector2f(150,60));
        sf::RectangleShape increase_n_button_rect   (sf::Vector2f(150,30));
        sf::RectangleShape decrease_n_button_rect   (sf::Vector2f(150,30));
        sf::RectangleShape increase_k_button_rect   (sf::Vector2f(150,30));
        sf::RectangleShape decrease_k_button_rect   (sf::Vector2f(150,30));

        start_button_rect.setFillColor              (sf::Color(200,200,200));
        stop_button_rect.setFillColor               (sf::Color(150,150,150));
        increase_n_button_rect.setFillColor         (sf::Color(100,100,100));
        decrease_n_button_rect.setFillColor         (sf::Color(175,175,175));
        increase_k_button_rect.setFillColor         (sf::Color(50,50,50));
        decrease_k_button_rect.setFillColor         (sf::Color(75,75,75));

        start_button_rect.setPosition       (0,600);
        stop_button_rect.setPosition        (150,600);
        increase_n_button_rect.setPosition  (300,600);
        decrease_n_button_rect.setPosition  (300,630);
        increase_k_button_rect.setPosition  (450,600);
        decrease_k_button_rect.setPosition  (450,630);

        buttons.push_back(std::pair<sf::RectangleShape, void (Game::*)(sf::Event)>
          (start_button_rect, Game::start));
        buttons.push_back(std::pair<sf::RectangleShape, void (Game::*)(sf::Event)>
          (stop_button_rect, Game::stop));
        buttons.push_back(std::pair<sf::RectangleShape, void (Game::*)(sf::Event)>
          (increase_n_button_rect, Game::inc_n));
        buttons.push_back(std::pair<sf::RectangleShape, void (Game::*)(sf::Event)>
          (decrease_n_button_rect, Game::dec_n));
        buttons.push_back(std::pair<sf::RectangleShape, void (Game::*)(sf::Event)>
          (increase_k_button_rect, Game::inc_k));
```

```
        buttons.push_back(std::pair<sf::RectangleShape, void (Game::*)(sf::Event)>
          (decrease_k_button_rect, Game::dec_k));

        R = new Ramsey(n,k); //Allocate memory for the Ramsey object on the heap.
    }

    void process(sf::Event event)   {
        if ( event.type == sf::Event::MouseButtonPressed) {
            for (auto b : buttons) {
                if(b.first.getGlobalBounds().contains(event.mouseButton.x,event.mouseButton.y)) {
                    (this->*b.second)(event);
                }
            }
        }
    }
    void update(int n = 8, int k = 8)  {
        R->update();  // Create a new random matrix
    }
    void draw(sf::RenderTarget& target, sf::RenderStates states) const; //Game has its own draw.
};

void Game::start(sf::Event e) {
   // Set R's is_valid to false, you need to make a new one.
}

void Game::stop(sf::Event e) {
    // Set R's is_valid to true, you have achieved Ramseyness.
}

void Game::inc_n(sf::Event e) {
    // Call R's adjust_n function with offset = 1.
}

void Game::dec_n(sf::Event e) {
    // Call R's adjust_n function with offset = -1.
}

void Game::inc_k(sf::Event e) {
    // Call R's adjust_k function with offset = 1.
}

void Game::dec_k(sf::Event e) {
    // Call R's adjust_n function with offset = -1.
}

void Game::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    for(auto b : buttons) target.draw(b.first);
  // Call R's draw function with parameters target and states.
}
```

And finally, the main():

```
#include "Game.hpp"

int main() {
    srand(time(0));
    sf::RenderWindow window(sf::VideoMode(600,660),"Ramsey Squares");

    window.setFramerateLimit(60);  // Throttle back the frame rate to 60.
    Game g(8,5); // The default game
    while(window.isOpen())  {
        sf::Event event;
        // Call g's update function
        while(window.pollEvent(event)) {
```

```
            // Call g's process function
            if(event.type == sf::Event::Closed)
                window.close();
        }
        window.clear();
        window.draw(g);
        window.display();
    }
    return 0;
}
```