**实验1-棋盘覆盖**

**问题分析**

* 当k=0时（1×1棋盘），即一个特殊方格，骨牌数为0；
* 当k=1时（2×2棋盘），存在一个特殊方格，用一个L型骨牌覆盖其他三个区域，骨牌数为1；
* 当k>1时

划分：将棋盘分割为4个的子棋盘。此时特殊方格位于4个子棋盘其中之一，而其余的3个子棋盘中没有特殊方格。在递归之前要将原问题转化为4个规模较小的相同子问题。回顾原问题为大棋盘中有一个特殊方格，那么子问题就是分割后的棋盘也有一个“特殊方格”。因此用一个L型骨牌覆盖其余3个子棋盘的汇合处，从而构造特殊方块。骨牌数为；

递归：递归地使用这种分割并构造L型骨牌。直到棋盘的宽度为1即只有一个方格，停止递归；

合并：此问题不涉及合并内容。

**关键函数**

**void** cb(**int** br, **int** bc, **int** tr, **int** tc,**int** size)

（br,bc）为当前棋盘的左上角坐标，(tr,tc)为特殊方格所在的坐，Size为当前棋盘的宽度（宽度为1时即为递归的终止条件）

对特殊方块位置的可用如下代码判断

1. **if** (br + n > tr && bc + n > tc )//左上角
2. **if** (br + n > tr && bc + n <= tc)//右上角
3. **if** (br + n <= tr && bc + n > tc)//左下角
4. **if** (br + n <= tr && bc + n <= tc)//右下角

以左上角为例，如果特殊方格在区域内，则直接递归调用cb函数。否则先填充特殊方块，后对其进行递归划分，此时cb函数中特殊方块的位置替换为当前填充的特殊方块的位置。填充方块的规则：左上区域填充右下角，右上区域填充左下角，左下区域填充右上角，右下区域填充左上角。

划分递归调用的顺序为左上、右上、左下、右下。

1. **if** (br + n > tr && bc + n > tc )
2. {
3. //在区域内
4. cb(br, bc, tr, tc, n);
5. }
6. **else**
7. {
8. //先填充特殊方块，后分割
9. board[br + n - 1][bc + n - 1] = c;
10. cb(br, bc, br + n -1, bc + n - 1, n);
11. }

**图形界面**

工具：visual studio SFML库

**前期工作准备**

* 给棋盘覆盖项目配置SFML库
* 构建基本的SFML窗口渲染循环。创建一个长宽均为800像素的SFML窗口
* 生成棋盘，使用SFML库中的矩形类并创建宽度和高度均为100像素的二维向量

1. sf::RenderWindow window(sf::VideoMode(800, 800), "Chessboard");
2. **while** (window.isOpen())
3. {
4. sf::Event event;
5. **while** (window.pollEvent(event))
6. {
7. **if** (event.type == sf::Event::Closed)
8. window.close();
9. }
10. window.clear();
11. // 绘制棋盘
12. sf::RectangleShape square(sf::Vector2f(100, 100));

**用不同颜色骨牌覆盖棋盘**

* 设计数字对应颜色函数，共有9种颜色选择，采用数字求余编码使得在小规模棋盘中不会出现相邻骨牌颜色重复的现象
* 遍历之前得到的board数组，根据函数得到相应颜色并对相应位置进行涂色

1. sf::Color int2color(**int** a)
2. {
3. **int** ans = a % 9;
4. **switch** (ans)
5. {
6. **case** 0:
7. **return** sf::Color::Red;
8. … …
9. **case** 8:
10. **return** sf::Color(255, 192, 203);//粉
11. }
12. }

使用数字对应颜色函数同时保证了相同骨牌颜色一致的情况。同时对于特殊方格采用黑色方格代替。

* 上述方法仍然在n的数量不大时可以避免骨牌颜色重复，但是当n增大仍有可能出现相同颜色相邻的情况。因此设计了一个随机颜色的功能，每个案例根据n的个数，随机初始化个颜色，后再根据board数组中存储的数据取出相应颜色

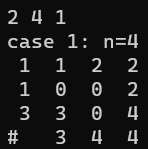
1. **int** num = (n \* n - 1) / 3 + 1;//骨牌数量
2. //初始化骨牌颜色
3. random\_device rd;
4. mt19937 gen(rd());
5. uniform\_int\_distribution<**int**> dis(0, 255);
6. vector<sf::Color> randomColors; // 存储随机颜色的容器
7. **for** (**int** i = 0; i < num; ++i)
8. {
9. sf::Color randomColor(dis(gen), dis(gen), dis(gen));
10. randomColors.push\_back(randomColor);
11. }

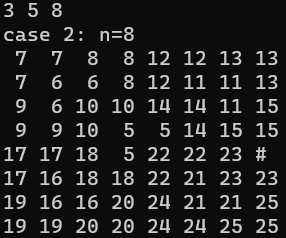
着色具体代码：

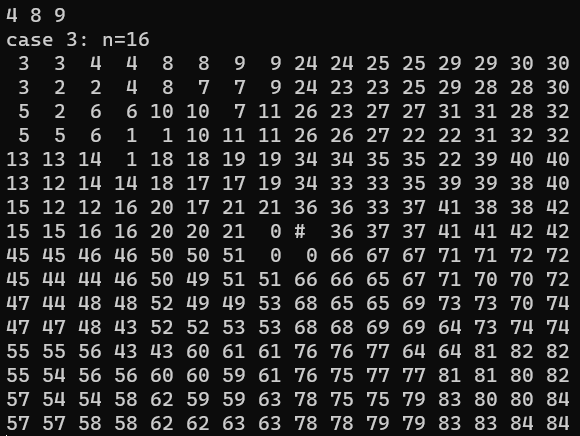
1. sf::RectangleShape square(sf::Vector2f(100, 100));
2. **for** (**int** i = 0; i < n; i++)
3. {
4. **for** (**int** j = 0; j < n; j++)
5. {
6. square.setFillColor(board[i][j] == -1 ? sf::Color::Black : randomColors[board[i][j]]);
7. square.setPosition(j \* 100, i \* 100);
8. window.draw(square);
9. }
10. }
11. window.display();

**运行结果截图**

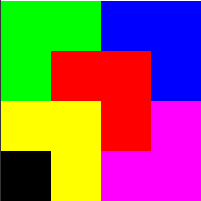
**数字形式输出**



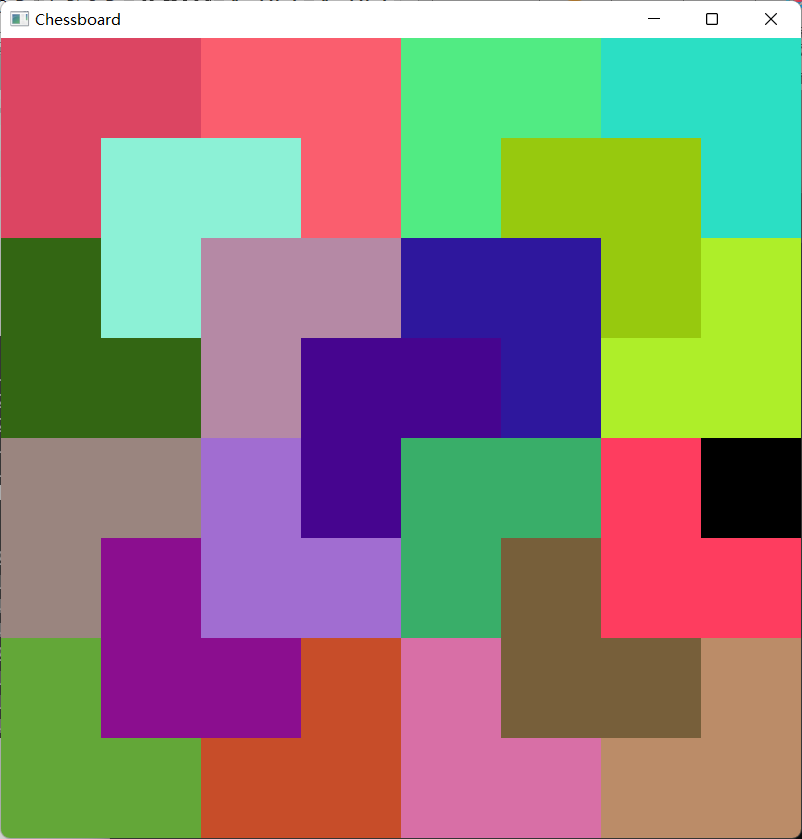




**图形化输出**



2 4 1



3 5 8



4 8 9

**设计调试中的问题**

* 忽略了递归终止的判断。解决方法在函数中判断size == 1;即当子问题仅为单个方格时直接返回。
* 初次使用SFML图形化界面发现最终出现的图像和目标图像呈对角线对称呈现。解决方法：上网查询了解到SFML的格式输入是square.setPosition(j \* 50, i \* 50);即先输入列后输入行。调换顺序后最终图像正确呈现。

**实验体会**

此次棋盘覆盖实验是算法设计与分析课程的第一次实验，虽然题目看起来并不复杂，但是对于之前没有算法基础的我来说还是有一定难度。这次实验让我第一次体会到看递归与分治算法的巧妙之处，它让看起来非常困难的问题变得易于处理，而其中关键点在于对于原问题的划分，如此只需解决一些类似的简单小规模子问题。在日后的算法学习过程中，我也会多多尝试递归与分治的思想去分析问题、解决问题。图形化界面的制作也让原本的数字化结果变得更直观更美观，同时图形化呈现也能使自身多了些许成就感。

通过这次实验让我将分治算法得以运用同时加深了我对递归策略的理解，并学会了如何在visual studio中配置SFML库，并通过代码去做一些简单的图形处理。

**程序源码**

1. #include <iostream>
2. #include <bits/stdc++.h>
3. #include <cmath>
4. #include <ctime>
5. #include <SFML/Graphics.hpp>
6. #include <random>
7. **using** **namespace** std;
8. **int** board[1024][1024];
9. **int** count = 0;
10. **void** cb(**int** br, **int** bc, **int** tr, **int** tc,**int** size)//br,bc棋盘左上角行列，tr,tc目标行列
11. {
12. **if** (size == 1)
13. **return**;
14. **int** c;
15. c = ::count++;//全局变量
16. **int** n = size / 2;
17. //左上角
18. **if** (br + n > tr && bc + n > tc )
19. {
20. //在区域内
21. cb(br, bc, tr, tc, n);
22. }
23. **else**
24. {
25. //先填充特殊方块，后分割
26. board[br + n - 1][bc + n - 1] = c;
27. cb(br, bc, br + n -1, bc + n - 1, n);
28. }
29. //右上角
30. **if** (br + n > tr && bc + n <= tc)
31. {
32. cb(br, bc + n, tr, tc, n);
33. }
34. **else**
35. {
36. board[br + n -1 ][bc + n] = c;
37. cb(br, bc+n, br + n - 1, bc + n, n);
38. }
39. //左下角
40. **if** (br + n <= tr && bc + n > tc)
41. {
42. cb(br + n, bc, tr, tc, n);
43. }
44. **else**
45. {
46. board[br + n][bc + n - 1] = c;
47. cb(br+n, bc, br + n, bc + n - 1, n);
48. }
49. //右下角
50. **if** (br + n <= tr && bc + n <= tc)
51. {
52. cb(br + n, bc + n, tr, tc, n);
53. }
54. **else**
55. {
56. board[br + n][bc + n ] = c;
57. cb(br + n, bc+n, br + n, bc + n, n);
58. }
59. }
61. **int** main()
62. {
63. **int** temp = 0;
64. **int** k, x, y;
66. random\_device rd;
67. mt19937 gen(rd());
68. uniform\_int\_distribution<**int**> dis(0, 255);//均匀分布
69. **while** (cin >> k >> x >> y)
70. {
71. ::count = 0;
72. vector<sf::Color> randomColors; // 存储随机颜色的容器
73. **int** n = pow(2, k);
74. **int** num = (n \* n - 1) / 3 + 1;//骨牌数量
75. //初始化骨牌颜色
77. randomColors.clear();
78. // 生成并保存随机颜色
79. **for** (**int** i = 0; i < num; ++i)
80. {
81. sf::Color randomColor(dis(gen), dis(gen), dis(gen));
82. randomColors.push\_back(randomColor);
83. }
84. temp++;
85. x--;
86. y--;
87. board[x][y] = -1;
88. cb(0, 0, x, y, n);
90. cout << "case " << temp << ": n=" << n << endl;
91. **for** (**int** i = 0; i < n; i++)
92. {
93. **for** (**int** j = 0; j < n; j++)
94. {
95. **if**(board[i][j] == -1)
96. cout <<setw(2) << "#  ";
97. **else**
98. cout << setw(2)<<board[i][j] << ' ';
99. }
100. cout << endl;
101. }
102. sf::RenderWindow window(sf::VideoMode(n\*50, n\*50), "Chessboard");
103. **while** (window.isOpen())
104. {
105. sf::Event event;
106. **while** (window.pollEvent(event))
107. {
108. **if** (event.type == sf::Event::Closed)
109. window.close();
110. }
111. window.clear();
112. // 绘制棋盘
113. sf::RectangleShape square(sf::Vector2f(50, 50));
114. **for** (**int** i = 0; i < n; i++)
115. {
116. **for** (**int** j = 0; j < n; j++)
117. {
118. square.setFillColor(board[i][j] == -1 ? sf::Color::Black : randomColors[board[i][j]]);
119. square.setPosition(j \* 50, i \* 50);
120. window.draw(square);
121. }
122. }
123. window.display();
124. }
125. }
126. **return** 0;
127. }