2019-2020 学年第一学期算法设计与分析

上机实验报告

实验名称：棋盘覆盖问题

学　号	秦敏浩	姓　名	17122490	评　分	
专　业	计算机科学与技术	实验类型	综合	任课教师	岳晓冬
完成日期	2019.10.4	实验学时	2		

一、实验问题描述

1、实验内容

棋盘覆盖问题：设 $n=2^k$（k≥0）。在一个 n×n 个方格组成的棋盘中，恰有 1 个方格与其他方格不同，称该方格为特殊方格。

现给定 k>1，$n=2^k$ 设计一个算法实现棋盘的一种覆盖。

2、实验目的

- 完成棋盘覆盖问题
- 理解分治策略的基本思路
- 学会计算分治算法的时间复杂度
- 尝试优化棋盘覆盖代码

3、实验环境

操作系统：Linux/Windows

编译环境：GNU G++14

二、算法设计与分析

1、算法基本思想

考虑到带一个特殊方格的 n=2 问题一定有解，考虑将 $n=2^k$ 的问题转化为 4 个 $n=2^{k-1}$ 分治求解。采用递归的方法解决问题，子问题大小为 2 时退出，否则设法将 4 个子区域都带上一个特殊方格。

问题的关键是将子问题划归为“带一个特殊方格”的区域。考虑到 4 个子区域中一定有一个已经包含特殊方格，而剩下 3 个必然包含一个“L”形的角落，必然可以构造出“4 个子区域都带上一个特殊方格”的条件，算法可行性由此得到保证。

2、算法设计与求解步骤

cur：当前已使用骨牌个数

ans：记录每一个位置对应骨牌编号

L：子问题左上角绝对横坐标

U：子问题左上角绝对纵坐标

n：子问题规模（半径）

x：特殊方格绝对横坐标

y：特殊方格绝对纵坐标

- 递归开始时，L=U=1,n 为最大规模，x 和 y 为输入特殊方格坐标。
- 每次递归采用上述思路将一个 n 的问题化为 4 个 n/2 的子问题。
- 每个子问题独立判断特殊方格是否在范围内，如在则直接递归，如不在则取靠近中心一角为特殊方格并赋值。
- 当 n=2 时结束递归，通过 cur 赋值 3 个方块，并将 cur 自增。题干保证初始条件 k>1。
- 最后输出方案。

3、算法分析

考虑到这是一个**分治问题**：

$$T(n)=\begin{cases} O(1) & n\le 2 \\ 4T\left(\frac{n}{2}\right)+O(1) & n>2 \end{cases}$$

Assume that $n=2^m$, then

$$\frac{T(2^m)}{4^m}=\frac{T(2^{m-1})}{4^{m-1}}+\left(\frac{1}{4}\right)^m$$

$$\Rightarrow \frac{T(2^m)}{4^m}=\sum_{i=2}^{m}\left(\frac{1}{4}\right)^i \simeq 1$$

$$\Rightarrow T(2^m)\simeq 4^m$$

$$\Rightarrow T(n)=O(n^2)$$

考虑到**每一个方格只会被访问和赋值一次**：

$$T(n)=O(n^2)$$

4、算法程序实现

篇幅原因，见附录或附件。

若老师有心，不妨留意结果分析中提出的优化问题。

三、调试与运行

输入：

2 2 3
3 5 3
4 8 10

输出：

```
2 2 3
Case 1: n=4
   2   2   4   4
   2   1   #   4
   3   1   1   5
   3   3   5   5

3 5 3
Case 2: n=8
   3   3   5   5  13  13  15  15
   3   2   2   5  13  12  12  15
   4   2   6   6  14  14  12  16
   4   4   6   1   1  14  16  16
   8   8   #  10   1  18  20  20
   8   7  10  10  18  18  17  20
   9   7   7  11  19  17  17  21
   9   9  11  11  19  19  21  21

4 8 10
Case 3: n=16
   4   4   6   6  14  14  16  16  46  46  48  48  56  56  58  58
   4   3   3   6  14  13  13  16  46  45  45  48  56  55  55  58
   5   3   7   7  15  15  13  17  47  45  49  49  57  57  55  59
   5   5   7   2   2  15  17  17  47  47  49  44  44  57  59  59
   9   9  11   2  19  19  21  21  51  51  53  53  44  61  63  63
   9   8  11  11  19  18  18  21  51  50  50  53  61  61  60  63
  10   8   8  12  20  18  22  22  52  52  50  54  62  60  60  64
  10  10  12  12  20  20  22   1  52   #  54  54  62  62  64  64
  25  25  27  27  35  35  37   1   1  67  69  69  77  77  79  79
  25  24  24  27  35  34  37  37  67  67  66  69  77  76  76  79
  26  24  28  28  36  34  34  38  68  66  66  70  78  78  76  80
  26  26  28  23  36  36  38  38  68  68  70  70  65  78  80  80
  30  30  32  23  23  40  42  42  72  72  74  65  65  82  84  84
  30  29  32  32  40  40  39  42  72  71  74  74  82  82  81  84
  31  29  29  33  41  39  39  43  73  71  71  75  83  81  81  85
  31  31  33  33  41  41  43  43  73  73  75  75  83  83  85  85
```

四、结果分析

理论分析和测试结果表面了实验的正确性。

值得注意的是，此处赋值顺序和书上代码是不一致的，这主要是由于本文实现时采用先赋值特殊方块后递归的方法，而书上代码恰好相反。两者都是可行的，且它们是本质相同的。

考虑到分治时 4 个子问题中**至少有 3 个是旋转后本质相同的**，可以**采用同一策略减少递归总数**，时间复杂度变为：

$$T(n)=\begin{cases}O(1) & n\le 2\\ 2T\left(\frac{n}{2}\right)+2\left(\frac{n}{2}\right)^2 & n>2\end{cases}$$

$$\text{Assume that } n=2^m, \text{then}$$

$$\frac{T(2^m)}{2^m}\simeq\frac{T(2^{m-1})}{2^{m-1}}+2^m$$

$$\Rightarrow\frac{T(2^m)}{2^m}=\sum_{i=2}^{m}2^i\simeq 2^m$$

$$\Rightarrow T(2^m)\simeq(2^m)^2$$

$$\Rightarrow T(n)=O(n^2)$$

总时间复杂度保持不变。但考虑到递归的调用过程，实战中可能出现常数级差距。由于代码会相对难写一些，且**主要时间花费在 IO** 上，该优化对此题意义不大。但对于其他分治问题，这种思路可能会带来可观的**时间复杂度优化**，如书中矩阵分治乘等。

五、本次实验的收获、心得体会

本次实验考察对分治思想的基本理解。代码不算特别复杂，但写完此题可以对分治思想有基本的理解。

然而不论如何，棋盘覆盖问题使用此思路求解只能算是奇技淫巧。只有当 n 为 2 的幂次时，此方法才能奏效。当棋盘大小为任意规模时，是否还有解？或是说，就拿目前的问题，问有多少本质不同的摆放方式？这些都不是使用此算法就可以解决的问题。

附录：代码实现（也可见附件）

```
#include <iostream>
#include <iomanip>
using namespace std;

int cur;
int ans[1025][1025];

// left, upper, scale, the position of the missing block
void dfs(int L,int U,int n,int x,int y){
    int tmp=++cur;
    if (n==2){ // sub problem is in a solvable scale
        for (int i=0;i<2;++i){
            for (int j=0;j<2;++j){
                if (L+i!=x||U+j!=y) ans[L+i][U+j]=tmp;
            }
        }
        return;
    }
    n/=2; // the scale of the sub problem is n/2
    if (x<L+n&&y<U+n)    //top left
        dfs(L,U,n,x,y);
    else{
        dfs(L,U,n,L+n-1,U+n-1);
        ans[L+n-1][U+n-1]=tmp;
    }
    if (x>=L+n&&y<U+n)   //top right
        dfs(L+n,U,n,x,y);
    else{
        dfs(L+n,U,n,L+n,U+n-1);
        ans[L+n][U+n-1]=tmp;
    }
    if (x<L+n&&y>=U+n)   // bottom left
        dfs(L,U+n,n,x,y);
    else{
        dfs(L,U+n,n,L+n-1,U+n);
        ans[L+n-1][U+n]=tmp;
    }
    if (x>=L+n&&y>=U+n) //bottom right
        dfs(L+n,U+n,n,x,y);
    else{
        dfs(L+n,U+n,n,L+n,U+n);
        ans[L+n][U+n]=tmp;
    }
```

```
}

int main(){
    int n,x,y,kase=0;
    while (cin>>n>>x>>y){
        n=(1<<n);
        cout<<"Case "<<++kase<<": n="<<n<<endl;
        ans[x][y]=0;
        cur=0;
        dfs(1,1,n,x,y);
        for (int i=1;i<=n;++i){
            for (int j=1;j<=n;++j){
                if (ans[i][j]==0){
                    cout<<"   #";
                }
                else{
                    cout<<setw(4)<<ans[i][j];
                }
            }
            cout<<endl;
        }
    }
    return 0;
}
```