

实验一：递归与分治

实验目的

理解递归算法的思想和递归程序的执行过程，并能熟练编写递归程序。

掌握分治算法的思想，对给定的问题能设计出分治算法予以解决。

实验预习内容

编程实现讲过的例题：二分搜索、合并排序、快速排序。

对本实验中的问题，设计出算法并编程实现。

实验内容和步骤

快速排序及第 k 小数

快速排序

问题分析及算法思路

快速排序的基本思想：

- 通过一趟排序将待排序列分割成**独立的两部分**，其中一部分的值比另一部分的小，则分别对两部分序列继续进行排序，达到整个序列有序

实现逻辑：

使用分治法将序列分成两个子序列

- 从序列中选出一个元素，称为基准值（pivot），即**确定分界点**，可以选取 $q[1]$, $q[(1+r)/2]$, $q[r]$ 或者随机
- 重新排序序列，所有元素小于基准值的放在基准值前面，所有元素比基准值大的摆在基准的后面（相同的数规定放到确定的某一边）。分区推出之后，基准处于中间位置，这一步称为**调整区间**
- 对于分出的两个子序列继续进行上述操作，**递归处理左右两段**

对于**调整区间**可实现的方法有：

- 开辟额外的数组 $a[]$ 、 $b[]$
- 扫描 $q[1:r]$
 - 当 $q[i] \leq x$ 时，将 x 插入到 $a[]$
 - 当 $q[i] \geq x$ 时，将 x 插入到 $b[]$

- 分别将a[],b[]中的数放在q中

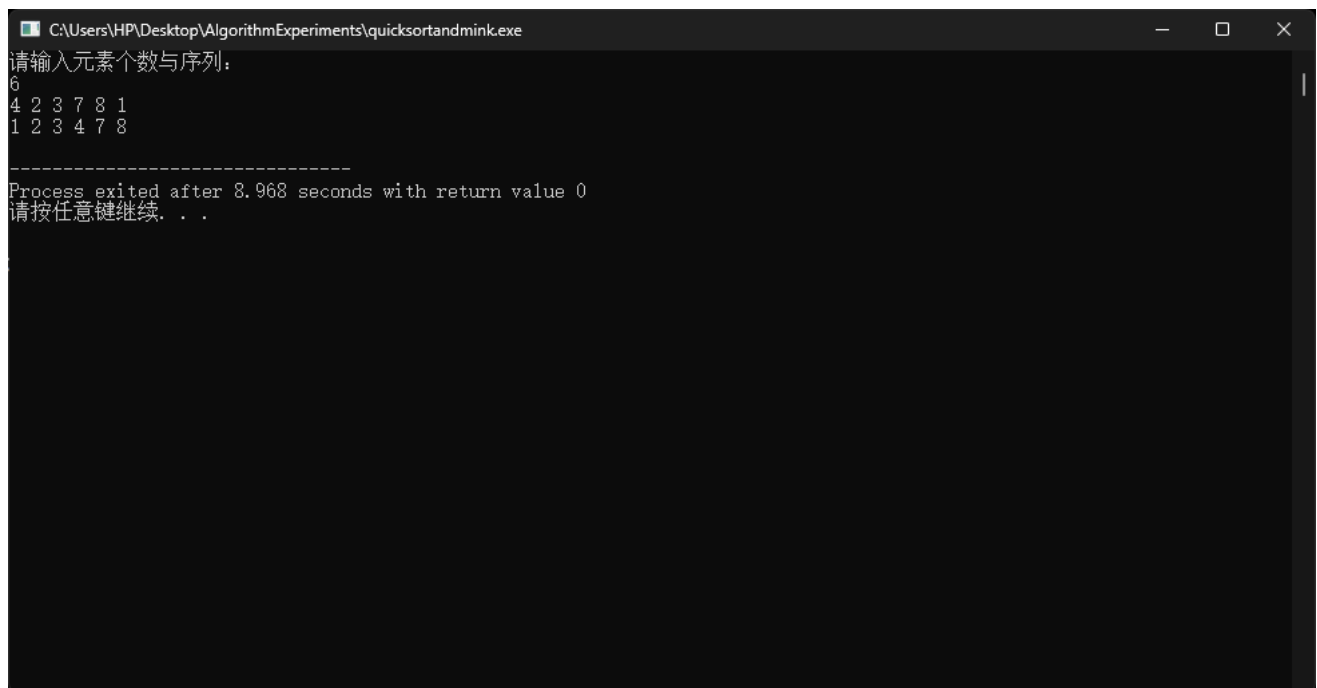
双指针法，设置i、j指针分别指向第一个和末尾一个，分别向左向右移动，知道重合

在本实验中，使用双指针法进行快排

算法设计与代码实现

```
void quicksort(int q[], int l, int r)
{
    if (l >= r) return;           //判边界
    int x = q[l], i = l - 1, j = r + 1;
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quicksort(q, l, j), quicksort(q, j + 1, r);
}
```

算法演示



```
C:\Users\HP\Desktop\AlgorithmExperiments\quicksortandmink.exe
请输入元素个数与序列:
6
4 2 3 7 8 1
1 2 3 4 7 8

-----
Process exited after 8.968 seconds with return value 0
请按任意键继续. . .
```

经过多次测试，算法的运行结果均正确，符合题意。

快速排序的算法时间复杂度是 $O(N * \log N)$

第k小数

这道题目是课外思考题top_k的一个变式，找到序列中第k小的数

问题分析及算法思路

基于快速排序的减治法

借用“快速排序”的思想，对整个序列进行**划分**(取序列第一个元素作为枢轴，也可采用随机法、三数取中法等方法)，并返回**划分后的位置**，若等于k则得到答案；若大于k，则说明该元素左边的元素都小于k，递归求解该位置前面序列第k小的元素即可；若小于k，则递归求解该位置后面序列第k - count(count为该序列中[l, j]的元素数量，其中j为划分后的元素位置)小的元素即可。由此，每次仅需求解大问题的一个子问题，最后即可得到第k小的数。

冒泡排序

使用冒泡排序的思想，每次冒泡都是找到了序列中的第i小的数（i为冒泡次数），i=k时找到第k小数

堆排序

堆排序中，每次弹出一个最小的值，只需要执行k次弹数操作就可以得到第k小数

堆

是一个完全二叉树

每个节点的值都大于或等于其叶子的值，为最大堆；反之为最小堆

堆排序

1. 将待排序的序列构造成一个最大堆，此时序列的最大值为根节点
2. 依次将根节点与待排序序列的最后一个元素交换
3. 再维护从根节点到该元素的前一个节点为最大堆，如此往复，得到一个递增序列

算法设计与代码实现

减治法

```
int quicksort(int a[], int l, int r)
{
    int pivot = a[l];
    int i = l - 1, j = r + 1;
    do
    {
        do { i++; } while (a[i] < pivot);
        do { j--; } while (a[j] > pivot);
        if (i < j) swap(a[i], a[j]);
    } while (i < j);

    return j;
}

void Top_k(int a[], int l, int r, int k)
{
    if (l >= r) return;
    int j = quicksort(a, l, r);
    int count = j - l + 1;
    if (count == k)
    {
        return;
    }
    else if (count > k)
    {
        Top_k(a, l, j, k);
    }
}
```

```

    }
    else
    {
        Top_k(a, j + 1, r, k - count);
    }
}

```

```

C:\Users\HP\Desktop\AlgorithmExperiments\program\mink_quicksort.exe
请输入元素个数与序列:
6
9 2 3 4 1 7
请输入要获得的第k小数
2
第2小数是2

-----
Process exited after 21.18 seconds with return value 0
请按任意键继续. . .

```

冒泡排序

```

void bubblesort(int a[], int k, int n)
{
    for (int i = 0; i < k; i++)
    {
        for (int j = n - 2; j >= i; j--)
        {
            if (a[j + 1] < a[j])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}

```

```
CAUsers\HP\Desktop\AlgorithmExperiments\program\mink_bubblesort.exe
请输入元素个数与序列:
6
9 2 3 4 1 7
请输入要获得的第k小数
2
2

-----
Process exited after 53.18 seconds with return value 0
请按任意键继续. . .
```

堆排序

```
void down(int u)
{
    int t = u;          // t为点、左孩子、右孩子三个点中最小的一个点
    if (u * 2 <= cnt && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= cnt && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t)          // 根节点不是最小的
    {
        // 与最小的点交换
        swap(h[u], h[t]);
        down(t);          // 递归处理
    }
}

void up(int u)
{
    while (u / 2 && h[u] < h[u / 2])
    {
        swap(h[u / 2], h[u]);
        u >>= 1;          // u /= 2换上去
    }
}

void HeapSort(int h[], int k, int n)
{
    for (int i = n / 2; i; i--) down(i);    // O(n)的时间复杂度建堆

    while (m--)
    {
        if (m == 0){
            cout<<h[1]<<endl;
        }
        h[1] = h[cnt--];
        down(1);
    }
}
```

```
C:\Users\HP\Desktop\AlgorithmExperiments\program\mink_heapsort.exe
6 2
9 2 3 4 1 7
2

-----
Process exited after 10.98 seconds with return value 0
请按任意键继续. . .
```

经过测试，上述方法均能实现查找第k小数的功能

关于上述方法的时间复杂度分析

- 对于减治法，时间复杂度是 $O(n)$
- 对于冒泡排序，时间复杂度是 $O(kn)$
- 对于堆排序，由于需要构建完整堆，时间复杂度是 $O(n * \log n)$

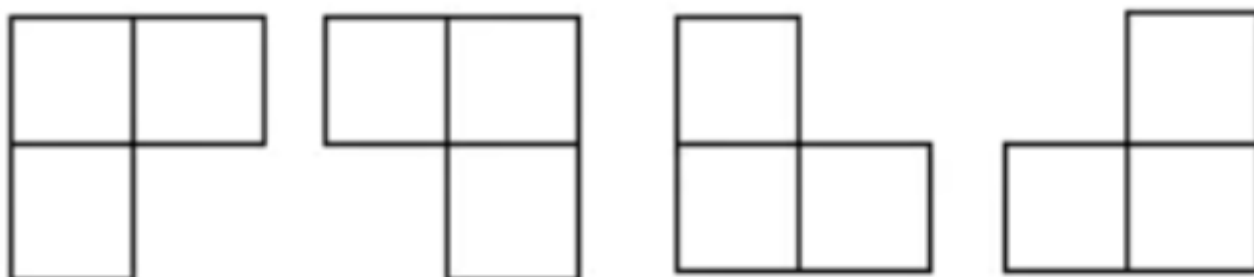
对于减治法，设每次划分的pivot恰好是序列的中值，可以保证每一次减治去掉一半的区间，由于一次划分耗费 $O(n)$ 的时间，因此只需要处理剩下的一半大小的子序列，得到递推公式

$$\begin{aligned}T(n) &= T(n/2) + O(n) \\T(n) &= O(n)\end{aligned}$$

棋盘覆盖问题

问题描述

在一个 $2^k * 2^k$ 方格组成的棋盘中，恰有一个方格与其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用图示的4种不同形态的L型骨牌覆盖给定的特殊棋盘上除特殊方格以外的所有方格，且任何2个L型骨牌不得重叠覆盖；

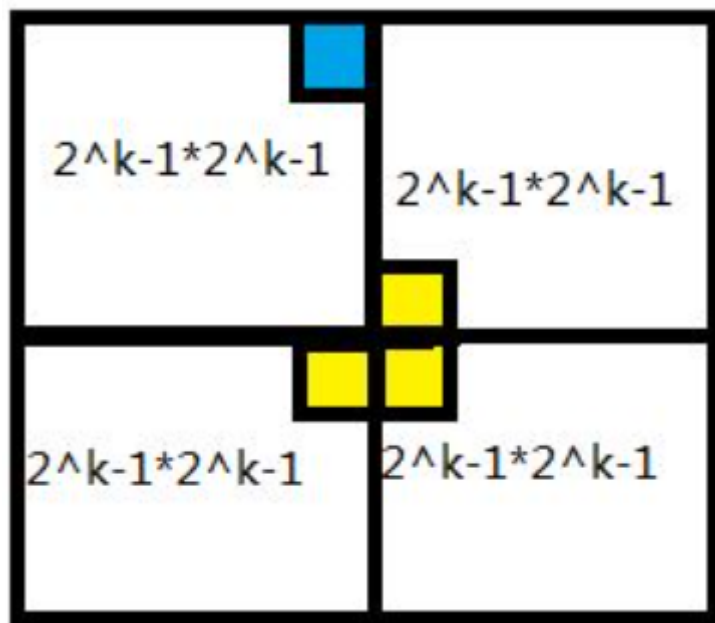


问题分析与算法思路

采用分治的思想，首先将棋盘进行划分：

1. 当 $k=0$ 时，是一个 1×1 的棋盘，棋盘中的骨牌数是0
2. 当 $k>0$ 时，将 $2^k \times 2^k$ 分割成4个 $2^{k-1} \times 2^{k-1}$ 的子棋盘，特殊方格必定位于4个子棋盘注意，其余3个子棋盘中没有特殊方格

对于划分出来的四个子棋盘，用一个L型骨牌覆盖3个没有特殊方格的子棋盘的连接处



至此，对每个棋盘按照左上、右上、右下、左下的顺时针顺序铺满棋盘；每次都对分割后的四个小方块进行特殊方格判断：

- 如果特殊方块在里面，递归
- 不在，根据分割的不同位置，把三个角落的方格标记为特殊方块，递归、

算法设计与代码实现

```
// 函数：棋盘覆盖
void chessboardCover(int tr, int tc, int dr, int dc, int size) {
    if (size == 1)
        return;

    int t = tileID++;
    int s = size / 2;

    // 左上角子棋盘
    if (dr < tr + s && dc < tc + s)
        chessboardCover(tr, tc, dr, dc, s);
    else {
        // 不在左上角，用t号骨牌覆盖右下角
        board[tr + s - 1][tc + s - 1] = t;
        // 覆盖其他方格
        chessboardCover(tr, tc, tr + s - 1, tc + s - 1, s);
    }

    // 右上角子棋盘
    if (dr < tr + s && dc >= tc + s)
        chessboardCover(tr, tc + s, dr, dc, s);
```

```

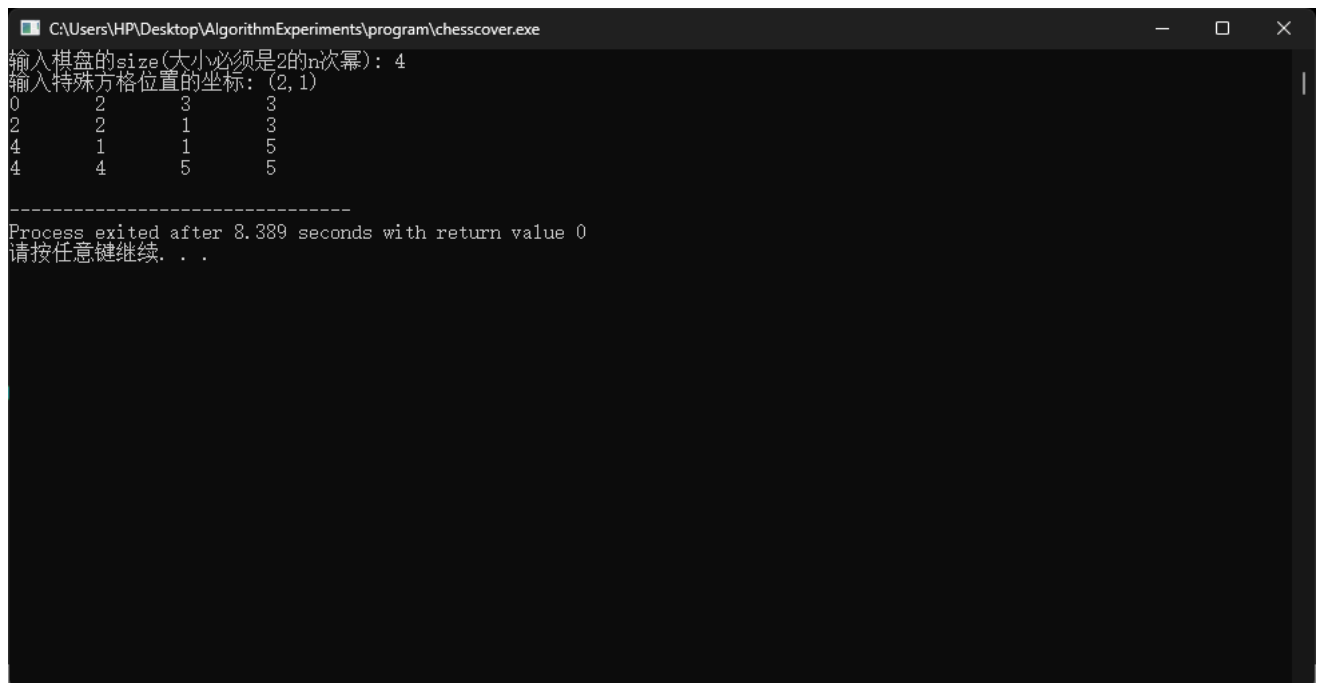
else {
    // 不在右上角，用t号骨牌覆盖左下角
    board[tr + s - 1][tc + s] = t;
    // 覆盖其他方格
    chessboardCover(tr, tc + s, tr + s - 1, tc + s, s);
}

// 左下角子棋盘
if (dr >= tr + s && dc < tc + s)
    chessboardCover(tr + s, tc, dr, dc, s);
else {
    // 不在左下角，用t号骨牌覆盖右上角
    board[tr + s][tc + s - 1] = t;
    // 覆盖其他方格
    chessboardCover(tr + s, tc, tr + s, tc + s - 1, s);
}

// 右下角子棋盘
if (dr >= tr + s && dc >= tc + s)
    chessboardCover(tr + s, tc + s, dr, dc, s);
else {
    // 不在右下角，用t号骨牌覆盖左上角
    board[tr + s][tc + s] = t;
    // 覆盖其他方格
    chessboardCover(tr + s, tc + s, tr + s, tc + s, s);
}
}

```

算法演示



```

C:\Users\HP\Desktop\AlgorithmExperiments\program\chesscover.exe
输入棋盘的大小(大小必须是2的n次幂): 4
输入特殊方格位置的坐标: (2, 1)
0      2      3      3
2      2      1      3
4      1      1      5
4      4      5      5

-----
Process exited after 8.389 seconds with return value 0
请按任意键继续. . .

```

经过多组测试，算法运行正确

算法分析

分析时间复杂度：

递推式

$$T(k) = \begin{cases} O(1) & k = 0 \\ 4T(k-1) + O(1) & k > 0 \end{cases}$$

解得

$$T(k) = O(4^k)$$

设 $n = 2^k$ ，得到时间复杂度是 $O(n^2)$

实验总结

分治即“**分而治之**”，一个规模很大的问题若要直接求解起来是非常困难的，将一个复杂的问题分解为若干个规模较小但是类似于原问题的子问题，子问题可以再分为更小的子问题，最终达到子问题可以简单的直接求解的目的，那么原问题的解即子问题的解的并集。分治算法可以缩小问题的规模，使得问题的求解变得十分容易。

附录

快速排序源代码

```
#include<iostream>
#include<algorithm>
#define N 20

using namespace std;

void quicksort(int q[], int l, int r)
{
    if (l >= r) return; //判边界
    int x = q[l], i = l - 1, j = r + 1;
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quicksort(q, l, j), quicksort(q, j + 1, r);
}

int main(){
    int n,q[N];
    cout<<"请输入元素个数与序列: "<<endl;
    cin>>n;
    for(int i=0;i<n;i++){
```

```

        cin>>q[i];
    }

    quicksort(q,0,n-1);

    for(int i=0;i<n;i++){
        cout<<q[i]<<" ";
    }

    cout<<endl;
    system("pause");
    return 0;
}

```

第 k 小数源代码

减治法

```

#include<iostream>
#include<algorithm>
using namespace std;
const int N = 20;

int quicksort(int a[], int l, int r)
{
    int pivot = a[l];
    int i = l - 1, j = r + 1;
    do
    {
        do { i++; } while (a[i] < pivot);
        do { j--; } while (a[j] > pivot);
        if (i < j) swap(a[i], a[j]);
    } while (i < j);

    return j;
}

void Top_k(int a[], int l, int r, int k)
{
    if (l >= r) return;
    int j = quicksort(a, l, r);
    int count = j - l + 1;
    if (count == k)
    {
        return;
    }
    else if(count > k)
    {
        Top_k(a, l, j, k);
    }
    else
    {

```

```

        Top_k(a, j + 1, r, k - count);
    }
}

int main()
{
    int a[N], k, n;
    cout << "请输入元素个数与序列:" << endl;
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    cout << "请输入要获得的第k小数" << endl;
    cin >> k;

    Top_k(a, 0, n - 1, k);

    cout << "第" << k << "小数是" << a[k - 1] << endl;
    cout << endl;
    return 0;
}

```

冒泡排序

```

#include<iostream>
using namespace std;
const int N = 20;

void bubblesort(int a[], int k, int n)
{
    for (int i = 0; i < k; i++)
    {
        for (int j = n - 2; j >= i; j--)
        {
            if (a[j + 1] < a[j])
            {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
            }
        }
    }
}

int main()
{
    int a[N], k, n;
    cout << "请输入元素个数与序列:" << endl;
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    cout << "请输入要获得的第k小数" << endl;
    cin >> k;
    bubblesort(a, k, n);
}

```

```

    cout << a[k - 1] << endl;
    return 0;
}

```

堆排序

```

#include<iostream>
#include<algorithm>
using namespace std;

const int N = 100010;

int n, m;
int h[N], cnt;

void down(int u)
{
    int t = u;          // t为点、左孩子、右孩子三个点中最小的一个点
    if (u * 2 <= cnt && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= cnt && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t)          // 根节点不是最小的
    {
        // 与最小的点交换
        swap(h[u], h[t]);
        down(t);          // 递归处理
    }
}

void up(int u)
{
    while (u / 2 && h[u] < h[u / 2])
    {
        swap(h[u / 2], h[u]);
        u >>= 1;          // u /= 2换上去
    }
}

void HeapSort(int h[], int k, int n)
{
    for (int i = n / 2; i; i--) down(i);    // O(n)的时间复杂度建堆

    while (m--)
    {
        if (m == 0){
            cout<<h[1]<<endl;
        }
        h[1] = h[cnt--];
        down(1);
    }
}

int main()
{
    cin>>n>>m;
    for (int i = 1; i <= n; i++) cin>>h[i];
    cnt = n;
    HeapSort(h, m, n);
}

```

```
    return 0;
}
```

棋盘覆盖问题源代码

```
#define _CRT_SECURE_NO_WARNINGS 1
#include<iostream>
using namespace std;
int num = 1;           //L型骨牌的编号(递增)
int board[100][100];  //棋盘
/*****
* tr--当前棋盘左上角的行号
* tc--当前棋盘左上角的列号
* dr--当前特殊方格所在的行号
* dc--当前特殊方格所在的列号
* size: 当前棋盘的:2^k
*****/
void chessboardCover(int tr, int tc, int dr, int dc, int size) {
    if (size == 1)
        return;

    int t = num++;
    int s = size / 2;

    // 左上角子棋盘
    if (dr < tr + s && dc < tc + s)
        chessboardCover(tr, tc, dr, dc, s);
    else {
        // 不在左上角, 用t号骨牌覆盖右下角
        board[tr + s - 1][tc + s - 1] = t;
        // 覆盖其他方格
        chessboardCover(tr, tc, tr + s - 1, tc + s - 1, s);
    }

    // 右上角子棋盘
    if (dr < tr + s && dc >= tc + s)
        chessboardCover(tr, tc + s, dr, dc, s);
    else {
        // 不在右上角, 用t号骨牌覆盖左下角
        board[tr + s - 1][tc + s] = t;
        // 覆盖其他方格
        chessboardCover(tr, tc + s, tr + s - 1, tc + s, s);
    }

    // 左下角子棋盘
    if (dr >= tr + s && dc < tc + s)
        chessboardCover(tr + s, tc, dr, dc, s);
    else {
        // 不在左下角, 用t号骨牌覆盖右上角
        board[tr + s][tc + s - 1] = t;
        // 覆盖其他方格
        chessboardCover(tr + s, tc, tr + s, tc + s - 1, s);
    }
}
```

```

// 右下角子棋盘
if (dr >= tr + s && dc >= tc + s)
    chessboardCover(tr + s, tc + s, dr, dc, s);
else {
    // 不在右下角，用t号骨牌覆盖左上角
    board[tr + s][tc + s] = t;
    // 覆盖其他方格
    chessboardCover(tr + s, tc + s, tr + s, tc + s, s);
}
}

int main()
{
    int size;
    cout << "输入棋盘的size(大小必须是2的n次幂): ";
    cin >> size;
    int index_x, index_y;
    cout << "输入特殊方格位置的坐标: ";
    getchar();
    cin >> index_x >> index_y;
    chessboardCover(0, 0, index_x - 1, index_y - 1, size);
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
            cout << board[i][j] << "\t";
        cout << endl;
    }
}

```