递归 — 基本思想

郭 炜　刘家瑛

北京大学

递归的基本思想

- 什么是递归
  - 递归 — 某个函数直接或间接的调用自身
  - 问题的求解过程

  →划分成许多相同性质的子问题的求解

  →而小问题的求解过程可以很容易的求出
  - 这些子问题的解就构成里原问题的解

递归的基本思想

- 总体思想
  - 待求解问题的解 → 输入变量x的函数f(x)
  - 通过寻找函数g( ), 使得f(x) = g(f(x-1))
  - 且已知f(0)的值, 就可以通过f(0)和g( )求出f(x)的值
- 推广
  - 扩展到多个输入变量*x*, *y*, *z*等, *x*-1也可以推广到 $x - x_1$,

只要递归朝着 “出口” 的方向即可

递归与枚举的区别

- 枚举:

  把一个问题划分成一组子问题, 依次对这些子问题求解

  - 子问题之间是横向的, 同类的关系

- 递归:

  把一个问题逐级分解成子问题

  - 子问题与原问题之间是纵向的, 同类的关系
  - 语法形式上: 在一个函数的运行过程中, 调用这个函数自己
    - 直接调用: 在fun()中直接执行fun()
    - 间接调用: 在fun1()中执行fun2(); 在fun2()中又执行fun1()

递归的三个要点

- 递归式:

  如何将原问题划分成子问题

- 递归出口:

  递归终止的条件, 即最小子问题的求解,可以允许多个出口

- 界函数:

  问题规模变化的函数, 它保证递归的规模向出口条件靠拢

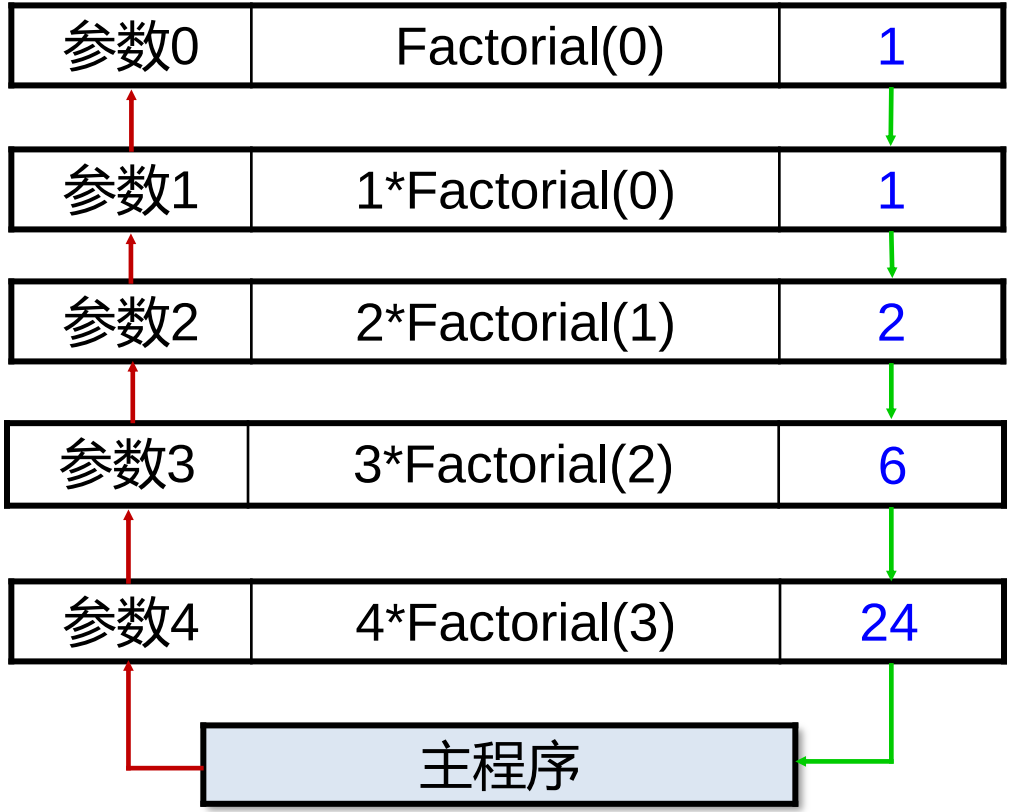
求阶乘的递归程序

- 给定n, 求阶乘n!

```
int n, m=1;
for (int i=2; i<=n; i++)
    m *= i;
printf(“%d的阶乘是%d\n”,
n, m);
```

```
int Factorial(int n){
    if (n == 0)
        return 1;
    else
        return  n * Factorial(n - 1);
}
```

阶乘的栈


参数0
Factorial(0)
1
参数1
1*Factorial(0)
1
参数2
2*Factorial(1)
2
参数3
3*Factorial(2)
6
参数4
4*Factorial(3)
24
主程序

递归解决问题的关键

1) 找出递推公式

2) 找到递归终止条件

注意事项: 由于函数的局部变量是存在栈上的
如果有体积大的局部变量, 比如数组,
而递归层次可能很深的情况下, 也许会导致栈溢出
→可以考虑使用全局数组或动态分配数组

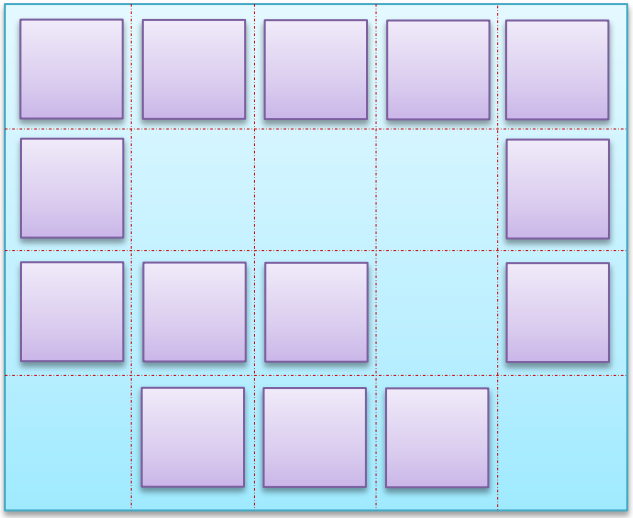
递归 — 小游戏

郭 炜　刘家瑛

北京大学

小游戏

- 问题描述
  - 一天早上, 你起床的时候想 “我编程序这么牛,
    为什么不能靠这个赚点小钱呢?”
    因此你决定编写一个小游戏
  - 游戏在一个分割成 w * h 个正方格子的矩形板上进行
  - 每个正方格子上可以有一张游戏卡片, 当然也可以没有

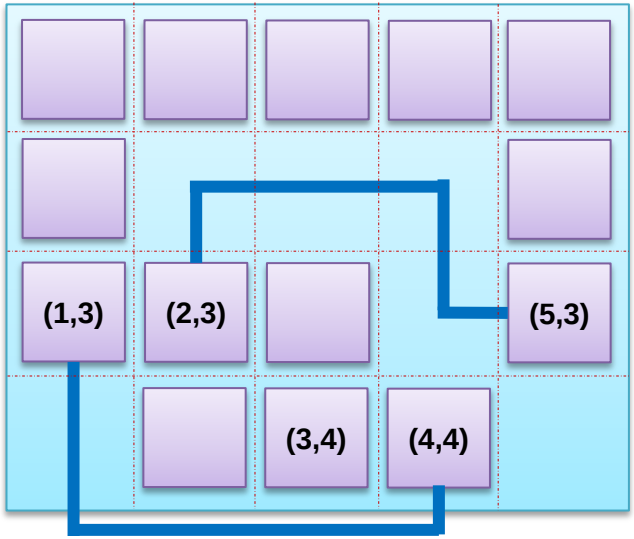


小游戏

- 问题描述
  - 当下面的情况满足时，

    认为两个游戏卡片之间有一条路径相连:
    - 路径只包含水平或者竖直的直线段
    - 路径不能穿过别的游戏卡片
    - 但是允许路径临时的离开矩形板

- 这是一个例子:



- 在 (1,3)和 (4,4)处的游戏卡片是可以相连的
- 而在 (2,3) 和 (3,4) 处的游戏卡是不相连的, 因为连接它们的每条路径都必须要穿过别的游戏卡片
- 现在要在小游戏里面判断:

  是否存在一条满足题意的路径能连接给定的两个游戏卡片

输入 (1/2)

- 输入包括多组数据: 一个矩形板对应一组数据
- 第一行包括两个整数 w和h (1 <= w, h <= 75),

分别表示矩形板的宽度和长度

- 下面的h行, 每行包括w个字符, 表示矩形板上的游戏卡片分布情况:
  - 使用 ‘X’ 表示这个地方有一个游戏卡片
  - 使用 空格 表示这个地方没有游戏卡片

输入 (2/2)

- 之后每行上包括4个整数:

  x1, y1, x2, y2 (1 <= x1, x2 <= w, 1 <= y1, y2 <= h)

- 给出两个卡片在矩形板上的位置

  注意: 矩形板左上角的坐标是(1,1)

  输入保证这两个游戏卡片所处的位置是不相同的

  如果一行上有4个0, 表示这组测试数据的结束

- 如果一行上给出w = h = 0, 那么表示所有的输入结束了

输出

- 对每一个矩形板, 输出一行 “Board #n:”, n是输入数据的编号
- 对每一组需要测试的游戏卡片输出一行. 这一行的开头是 “Pair m: ”, 这里m是测试卡片的编号（对每个矩形板, 编号都从1开始）
- 如果可以相连, 找到连接这两个卡片的所有路径中包括线段数最少的路径, 输出 “k segments.”

  k是找到的最优路径中包括的线段的数目
- 如果不能相连, 输出 “impossible.”
- 每组数据之后输出一个空行

样例输入

```
5    4
X X X X X
X          X
X X X    X
   X X X
2    3    5    3
1    3    4    4
2    3    3    4
0    0    0    0
0    0
```

样例输出

Board #1:

Pair 1: 4 segments.

Pair 2: 3 segments.

Pair 3: impossible.

问题分析 (1)

- 迷宫求解问题

  自相似性表现在每走一步的探测方式相同,

  可以用递归方法求解

- 通过枚举方式找到从起点到终点的路径, 朝一个方向走下去:
  - 如果走不通, 则换个方向走

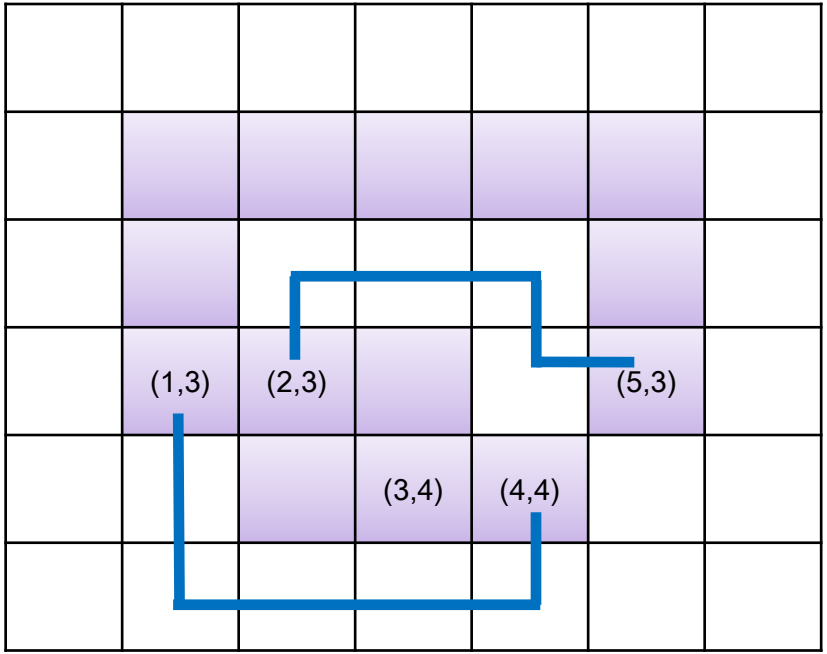
  →四个方向都走不通, 则回到上一步的地方, 换个方向走

  →依次走下去, 直到走到终点

问题分析 (1)

- 计算路径数目:
- 普通迷宫问题的路径数目是经过的格子数目
- 而该问题路径只包含水平或者竖直的直线段,
- 所以需要记录每一步走的方向
  - 如果上一步走的方向和这一步走的方向相同, 递归搜索时路径数不变, 否则路径数加1

问题分析 (2)



- 路径只包含水平或者竖直的直线段. 路径不能穿过别的游戏卡片. 但是允许路径临时的离开矩形板
- 所以在矩形板最外层增加一圈格子, 路径可以通过这些新增加的格子

问题分析 (3)

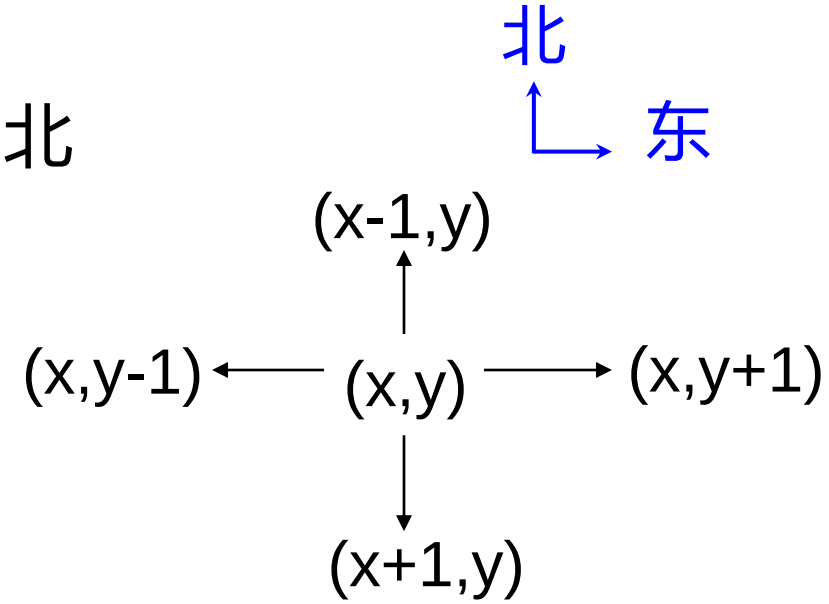
- 描述迷宫:

1. 设置迷宫为二维数组board[][], 数组的值是:
   空格: 代表这个地方没有游戏卡片
   'X' : 代表这个地方有游戏卡片
2. 在搜索过程中, 用另外一个二维数mark[][]标记格子是否已经走过了
   mark[i][j]=0 //格子(i, j)未走过
   mark[i][j]=1 //格子(i, j)已经走过
3. int minstep, w, h; //全局变量
   //minstep, 记录从起点到达终点最少路径数,
   //初始化为一个很大的数
   //w, h矩形板的宽度和高度

问题分析 (4)

- 设置搜索方向顺序是东, 南, 西, 北

```
int to[4][2] = {{0,1},{1,0},{0,-1},{-1,0}};
//now_x, now_y, 当前位置
//x, y下一步位置
for(i = 0; i < 4; i ++){
    int x = now_x + to[i][0];
    int y = now_y + to[i][1];
    f=i;  //方向, 0,1,2,3分别表示东,南,西,北
    …
}
```



问题分析 (5)

- 判断新位置(x, y)是否有效
  - T1: (x, y)在边界之内
    (x > -1) && (x < w + 2) && (y > -1) && (y < h + 2)
  - T2: 该位置没有游戏卡片并且未曾走过
    ((board[y][x] == ' ') && (mark[y][x] == false))
  - T3: 已经到达终点
    (x == end_x) && (y == end_y) && (board[y][x] == 'X‘)

综上, (x,y)有效的条件是 T1 && (T2 || T3)

```
((x > -1) && (x < w + 2) && (y > -1) && (y < h + 2)
&& (((board[y][x] == ' ') && (mark[y][x] == false))
|| ((x == end_x) && (y == end_y) && (board[y][x] == 'X'))))
```

递归方法

- 构造递归函数

```
void Search(int now_x, int now_y, int end_x, int end_y, int step, int f) ;
        //now_x, now_y当前位置
        //end_x, end_y结束位置
        //step已经走过的路径数目
        //f从上一步走到(now_x, now_y)时的方向
```

参考程序

```
#include <stdio.h>
#include <memory.h>
#define MAXIN 75
char board[MAXIN + 2][MAXIN + 2];    //定义矩形板
int minstep, w, h, to[4][2] = {{0,1},{1,0},{0,-1},{-1,0}};    //定义方向
bool mark[MAXIN + 2][MAXIN + 2];    //定义标记数组
```

```
void Search(int now_x, int now_y, int end_x, int end_y, int step, int f){
    if(step > minstep) return;  //当前路径数大于minstep, 返回→优化策略
    if(now_x == end_x && now_y == end_y){   //到达终点
        if(minstep > step)    //更新最小路径数
            minstep = step;
            return;
        }
```

```
    for(int i = 0; i < 4; i ++){    //枚举下一步的方向
            int x = now_x + to[i][0];   //得到新的位置
            int y = now_y + to[i][1];
            if ((x > -1) && (x < w + 2) && (y > -1) && (y < h + 2)
            && (((board[y][x] == ' ') && (mark[y][x] == false))||((x==end_x)
            && (y == end_y) && (board[y][x] == 'X'))))){
                mark[y][x] = true; //如果新位置有效标记该位置
                //已经过上一步方向和当前方向相同,
                //则递归搜索时step不变, 否则step+1
                if(f == i)  Search(x, y, end_x, end_y, step, i);
                else        Search(x, y, end_x, end_y, step + 1, i);
                mark[y][x] = false;    //回溯, 该位置未曾走过
            }
        }
}
```

```
int main(){
    int Boardnum = 0;
    while(scanf(“%d %d”, &w, &h)){    //读入数据
      if(w == 0 && h == 0)break;
      Boardnum ++;
      printf("Board #%d:\n", Boardnum);
      int i, j;
      for (i = 0; i < MAXIN + 2; i ++)board[0][i] = board[i][0] = ' ';
      for(i = 1; i <= h; i ++){    //读入矩形板的布局
            getchar();
            for(j = 1; j <= w; j ++) board[i][j] = getchar();
      }
      //在矩形板最外层增加一圈格子
      for (i = 0; i <= w; i ++)
            board[h + 1][i + 1] = ' ';
      for (i = 0; i <= h; i ++)
            board[i + 1][w + 1] = ' ';
```

```
    int begin_x, begin_y, end_x, end_y, count = 0;
    while(scanf("%d %d %d %d", &begin_x, &begin_y, &end_x, &end_y)
    && begin_x > 0){   //读入起点和终点
        count ++;
        minstep = 100000;    //初始化minstep为一个很大的值
        memset(mark, false, sizeof(mark));
        //递归搜索
        Search(begin_x, begin_y, end_x, end_y, 0, -1);
        //输出结果
        if(minstep < 100000)printf("Pair %d: %d segments.\n", count, minstep);
        else printf("Pair %d: impossible.\n", count);
     }
     printf("\n");
  }
  return 0;
}
```

问题小结

- 递归的条件
  - 自相似性表现在每走一步的探测方式相同, 可以用递归算法求解
- 定义并记录路径方向
- 判断下一步的位置是否符合要求
- 搜索过程Search( )
  - 朝一个方向走下去, 如果走不通, 则换个方向走; 四个方向都走不通, 则回到上一步的地方, 换个方向走; 依次走下去, 直到走到终点
- 计算路径数目
  - 需要记录每一步走的方向, 如果上一步走的方向和这一步走的方向相同, 递归搜索时路径数不变, 否则路径数加1

递归 — 棋盘分割

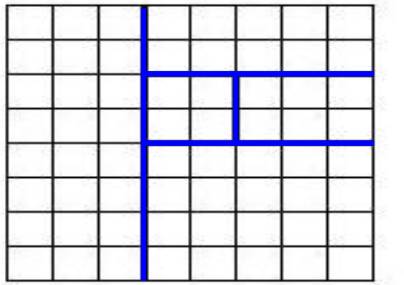
郭 炜　刘家瑛

北京大学

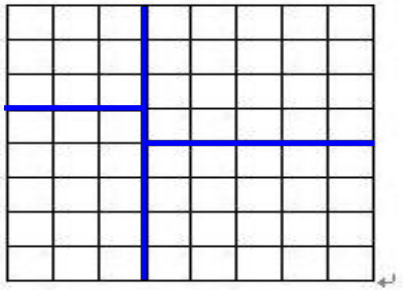
棋盘分割

- 将一个8*8的棋盘进行如下分割:
- 将原棋盘割下一块矩形棋盘并使剩下部分也是矩形,
- 再将剩下的部分继续如此分割, 这样割了(n-1)次后,
- 连同最后剩下的矩形棋盘共有n块矩形棋盘.

(每次切割都只能沿着棋盘格子的边进行)



允许的分割方案



不允许的分割方案

- 原棋盘上每一格有一个分值，
  一块矩形棋盘的总分为其所含各格分值之和
- 现在需要把棋盘按上述规则分割成 n 块矩形棋盘，
  并使各矩形棋盘总分的均方差最小

均方差 $\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n}}$，其中平均值 $\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$，

x_i 为第 i 块矩形棋盘的总分

请编程对给出的棋盘及 n，求出 σ 的最小值

- 输入

  第1行为一个整数n (1 < n < 15)
  第2行至第9行每行为8个小于100的非负整数, 表示棋盘上相应格子的分值

  每行相邻两数之间用一个空格分隔

- 输出

  仅一个数, 为σ（四舍五入精确到小数点后三位）

样例输入

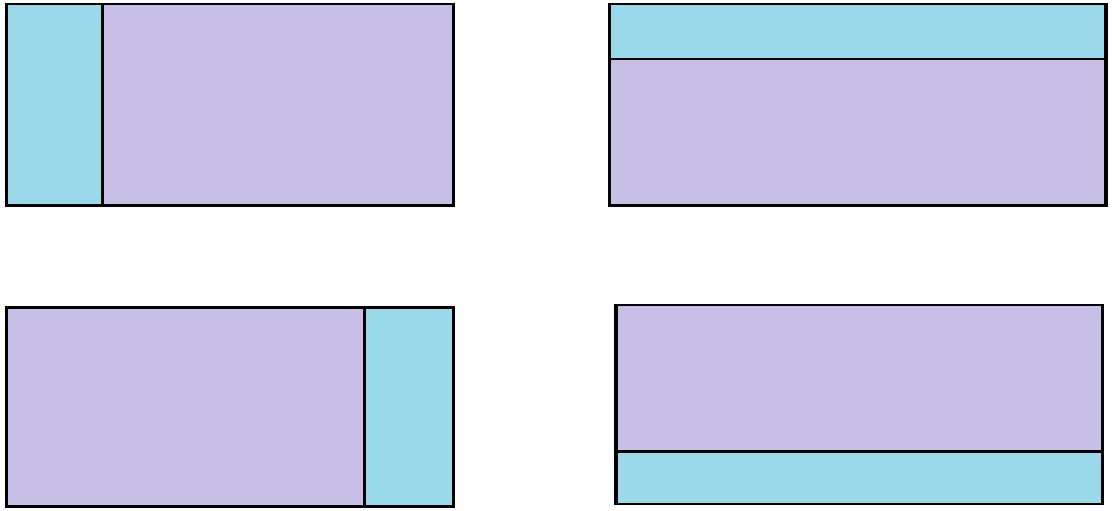
```
3
1 1 1 1 1 1 1 3
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0
1 1 1 1 1 1 0 3
```

样例输出

```
1.633
```

问题分析 (1)

- 每一次分割有以下4种方法:



f (k, 棋盘) = { f (1, 割下的棋盘) + f (k-1, 待割的棋盘)} (k≥2)

问题分析 (2)

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}}$$

- 如右式, 若要求出最小方差, 只需要求出最小的$\sum x_i^2$

$$\begin{aligned}
&\sum (x_i - \bar{x})^2 \\
&= \sum (x_i^2 - 2x_i\bar{x} + \bar{x}^2) \\
&= \sum x_i^2 - \sum 2x_i\bar{x} + n\bar{x}^2 \\
&= \sum x_i^2 - 2n\bar{x}^2 + n\bar{x}^2 \\
&= \sum x_i^2 - n\bar{x}^2
\end{aligned}$$

问题分析 (3)

- 设fun(n,x1,y1,x2,y2)为以(x1, y1)为左上角, (x2, y2)为右下角的棋盘分割成n份后的最小平方和
- 那么fun(n,x1,y1,x2,y2)=

$$\min\{\min_{i=x1}^{x2-1}\{fun(n-1,x1,y1,i,y2)+fun(1,i+1,y1,x2,y2)\},$$
$$\min_{i=x1}^{x2-1}\{fun(1,x1,y1,i,y2)+fun(n-1,i+1,y1,x2,y2)\},$$
$$\min_{i=y1}^{y2-1}\{fun(n-1,x1,y1,x2,i)+fun(1,x1,i+1,x2,y2)\},$$
$$\min_{i=y1}^{y2-1}\{fun(1,x1,y1,x2,i)+fun(n-1,x1,i+1,x2,y2)\}\}$$

其中fun(1,x1,y1,x2,y2)等于该棋盘内分数和的平方

问题分析 (3)

- 只想到这个还不够, TLE!
- 对于某个fun(n,x1,y1,x2,y2)来说, 可能使用多次这个值, 所以每次都计算太消耗时间
- 解决办法: 记录表
  - 用res[n][x1][y1][x2][y2]来记录fun(n,x1,y1,x2,y2)
  - res初始值统一为-1
  - 当需要使用fun(n,x1,y1,x2,y2)时, 查看res[n][x1][y1][x2][y2]
    - 如果为-1, 那么计算fun(n,x1,y1,x2,y2), 并保存于res[n][x1][y1][x2][y2]
    - 如果不为-1, 直接返回res[n][x1][y1][x2][y2]

参考程序

```
int s[9][9];    //每个格子的分数
int sum[9][9];    //(1,1)到(i,j)的矩形的分数之和
int res[15][9][9][9][9];    //fun的记录表

int calSum(int x1,int y1,int x2,int y2)//(x1,y1)到(x2,y2)的矩形的分数之和
{
    return sum[x2][y2]-sum[x2][y1-1]-sum[x1-1][y2]+sum[x1-1][y1-1];
}
```

```
int fun(int n,int x1,int y1,int x2,int y2)
{
    int t, a, b, c, e, MIN=10000000;
    if(res[n][x1][y1][x2][y2] !=-1)
        return res[n][x1][y1][x2][y2];
    if(n==1) {
        t=calSum(x1,y1,x2,y2);
        res[n][x1][y1][x2][y2]=t*t;
        return t*t;
     }
```

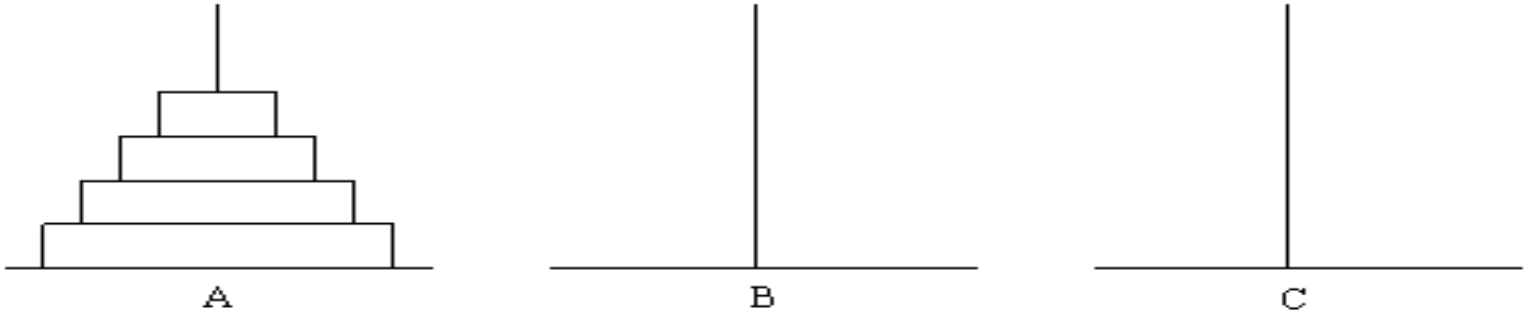
```
    for(a=x1;a<x2;a++) {
        c=calSum(a+1,y1,x2,y2);
        e=calSum(x1,y1,a,y2);
        t=min(fun(n-1,x1,y1,a,y2)+c*c, fun(n-1,a+1,y1,x2,y2)+e*e);
        if(MIN>t) MIN=t;
    }
    for(b=y1;b<y2;b++) {
        c=calSum(x1,b+1,x2,y2);
        e=calSum(x1,y1,x2,b);
        t=min(fun(n-1,x1,y1,x2,b)+c*c, fun(n-1,x1,b+1,x2,y2)+e*e);
        if(MIN>t) MIN=t;
    }
    res[n][x1][y1][x2][y2]=MIN;
    return MIN;
}
```

```
int main() {
    memset(sum, 0, sizeof(sum));
    memset(res, -1, sizeof(res));   //初始化记录表
    int n;
    cin>>n;
    for (int i=1; i<9; i++)
        for (int j=1, rowsum=0; j<9; j++) {
            cin>>s[i][j];
            rowsum +=s[i][j];
            sum[i][j] += sum[i-1][j] + rowsum;
    }
    double result = n*fun(n,1,1,8,8)-sum[8][8]*sum[8][8];
    cout<<setiosflags(ios::fixed)<<setprecision(3)<<sqrt(result/(n*n))<<endl;
    return 0;
}
```

用栈替代递归

汉诺塔问题

古代有一个梵塔，塔内有三个座A、B、C，A座上有64个盘子，盘子大小不等，大的在下，小的在上（如图）。有一个和尚想把这64个盘子从A座移到B座，但每次只能允许移动一个盘子，并且在移动过程中，3个座上的盘子始终保持大盘在下，小盘在上。在移动过程中可以利用B座，要求输出移动的步骤。



汉诺塔问题递归解法

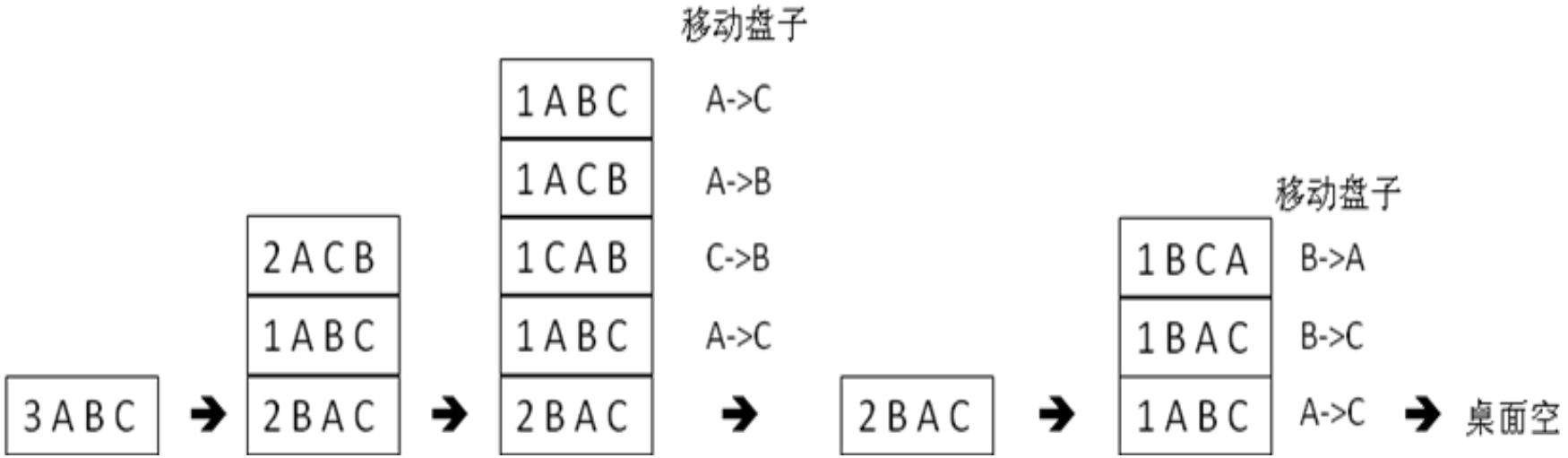
```
#include <iostream>
using namespace std;
void Hanoi(int n, char src,char mid,char dest)
//将src座上的n个盘子，以mid座为中转，移动到dest座
{
        if( n == 1) {  //只需移动一个盘子
            cout << src << "->" << dest << endl; //直接将盘子从src移动到dest即可
           return ;   //递归终止
        }
        Hanoi(n-1,src,dest,mid); //先将n-1个盘子从src移动到mid
        cout << src << "->" << dest << endl; //再将一个盘子从src移动到dest
        Hanoi(n-1,mid,src,dest); //最后将n-1个盘子从mid移动到dest
    return ;
}
```

汉诺塔问题递归解法

```
int main()
{
        int n;
        cin >> n; //输入盘子数目
        Hanoi(n,'A','B','C');
        return 0;
}
```

汉诺塔问题手工解法(三个盘子）

信封堆,每个信封放一个待解决的问题



汉诺塔问题非递归解法

```
#include <iostream>
#include <stack>
using namespace std;
struct Problem  {
        int n;
        char src,mid,dest;
        Problem( int nn, char s,char m,char d) :n(nn),src(s),mid(m),dest(d) {  }
}; //一个Problem变量代表一个子问题，将src上的n个盘子，
  // 以mid为中介，移动到dest
stack<Problem> stk;          //用来模拟信封堆的栈，一个元素代表一个信封
                             //若有n个盘子，则栈的高度不超过 n * 3
```

```
int main()  {
   int n;    cin >> n;
   stk.push(Problem(n,'A','B','C')); //初始化了第一个信封
   while( ! stk.empty()) { //只要还有信封，就继续处理
              Problem curPrb = stk.top(); //取最上面的信封，即当前问题
              stk.pop();  // 丢弃最上面的信封
               if( curPrb.n == 1 )   cout << curPrb.src << "->" << curPrb.dest << endl ;
                else {  //分解子问题
                        //先把分解得到的第3个子问题放入栈中
                       stk.push(Problem(curPrb.n -1,curPrb.mid,curPrb.src,curPrb.dest));
                       //再把第2个子问题放入栈中
                       stk.push(Problem(1,curPrb.src,curPrb.mid,curPrb.dest));
                       //最后放第1个子问题，后放入栈的子问题先被处理
                       stk.push(Problem(curPrb.n -1,curPrb.src,curPrb.dest,curPrb.mid));
              }
    }
   return 0;
}
```

汉诺塔问题递归解法

编译器生成的代码自动维护一个问题的栈，相当于信封堆。栈里每个子问题的描述中多了一项 --- 返回地址，返回地址可以描述该子问题已经解决到哪个步骤了，下面的(0) (1),(2),(3)就是返回地址

```
void Hanoi(int n, char src,char mid,char dest)
{
        if( n == 1) {
         (0)cout << src << "->" << dest << endl;
            return ;
        }
        Hanoi(n-1,src,dest,mid);
        (2)cout << src << "->" << dest << endl;
        Hanoi(n-1,mid,src,dest);
        (3)return ;
}
```

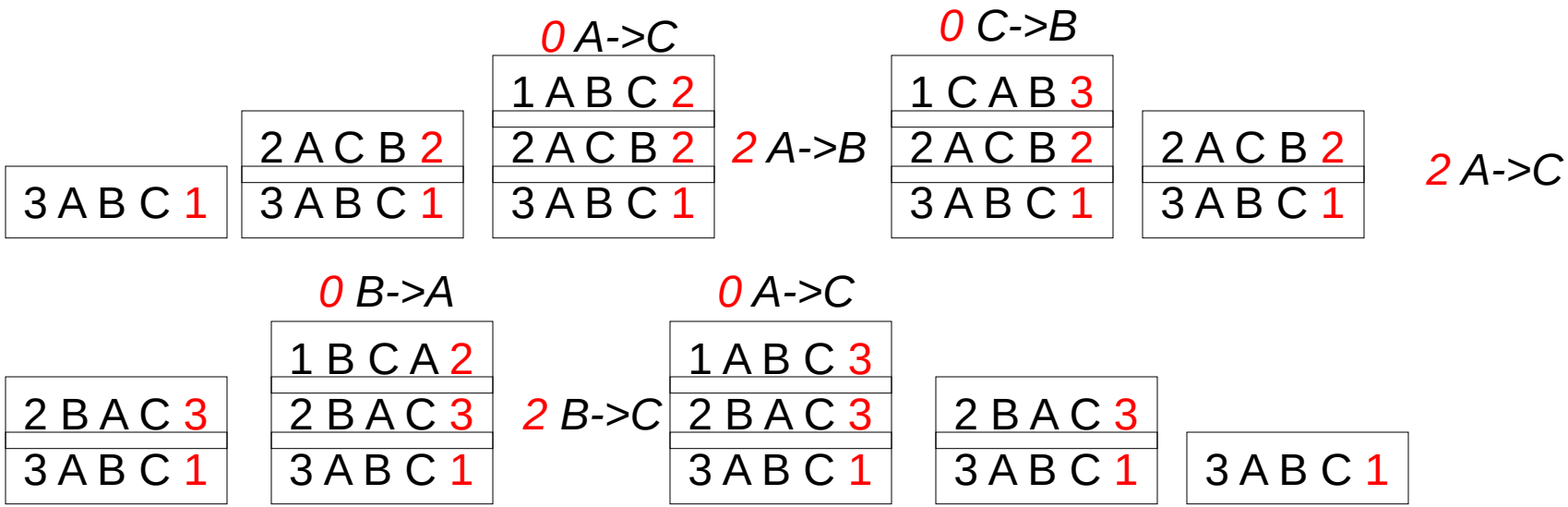
```
int main() {
        int n;
        cin >> n;
        Hanoi(n,'A','B','C');
        (1) return 0;
}
```

main中调用Hanoi时，栈形成初始状态：

n	A	B	C	1

汉诺塔问题递归解法

n = 3时，栈的变化状态:




```
void Hanoi(int n, char src,char mid,char dest)  {
                if( n == 1) {     (0)cout << src << "->" << dest << endl;       return ;       }
                Hanoi(n-1,src,dest,mid);
                (2)cout << src << "->" << dest << endl;
                Hanoi(n-1,mid,src,dest);
                (3)return ;
}
```

```
int main() {
    int n;
    cin >> n;    Hanoi(n,'A','B','C');
    (1) return 0;
}
```