

2024年春

# 程序设计实习(II): 算法设计

## 第十五讲 递 归

贾川民  
北京大学



# 递归的基本思想

## ■ 什么是递归

### □ 递归

- 某个函数直接/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
- 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  $\rightarrow$  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

# 递归和枚举的区别

## ■ 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3个要素

## 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

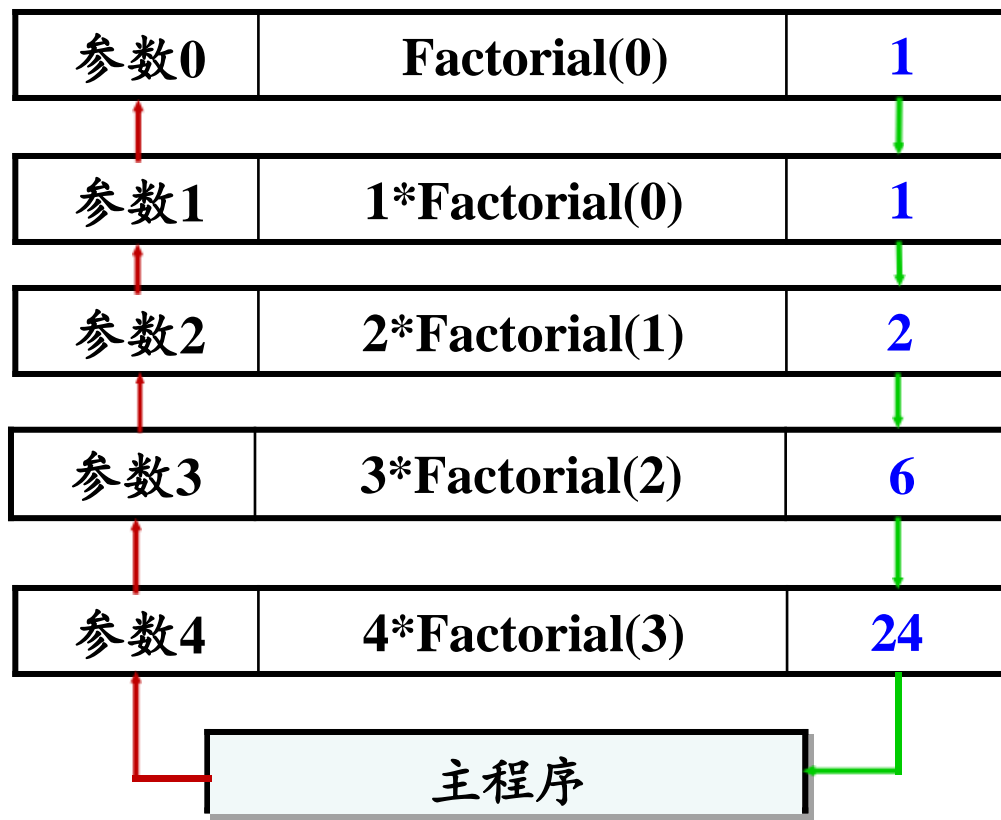
给定n, 求解n的阶乘  $n!$

```
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);
}
```

# 阶乘的程序栈 (Stack) 调用关系



# 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，  
而递归层次又可能很深的情况下，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**



# 递归求解的效率问题

- 递归求解时, 会将原问题分解为一个或者多个同类的子问题
- 若原问题被分解成A和B两个子问题
  - A被进一步分解为 $A_1$ 和 $A_2$ 两个子问题
  - B被进一步分解为 $B_1$ 和 $B_2$ 两个子问题
  - $A_1$ 可能与 $B_1$ 和 $B_2$ 的某一个完全相同
- 应通过**递归结果的存储**, 避免递归过程中的重复计算

## ■ 问题描述

国王每天向他的武士支付金币

国王确定每天所付金币数量的规则是：

- 第1天，国王付给武士1枚金币

- 接下来的2天，国王每天付给武士2枚金币

- 再接下来的3天，国王每天付给武士3枚金币

- .....

- 从第 $1+2+ \dots +(N-1)$ 天之后的连续N天，每天付给武士的金币数均为N枚

■ 请问：第y天结束时，武士共获得了多少枚金币

## ■ 输入

- 至少1行，不超过21行。每行一个整数
- 最后一行的整数为0，表示输入结束
- 其他行分别是[1 10000]区间内的某个整数 $y$ ，表示要计算第 $y$ 天时武士共获得了多少枚金币

## ■ 输出

- 除最后一行输入外，每行输入对应一行输出，包括两个整数。两个整数之间用一个空格分隔
  - 第一个整数是所输入的整数 $y$
  - 第二个整数是第 $y$ 天时武士获得的金币总数

得金币数    天数

**1**

**1**

**2**

**2**

**3**

**3**

**4**

**5**

**6**

**4**

**7**

**8**

**9**

**10**

**5**

**11**

**12**

**13**

**14**

**15**

.....

## 样例输入

10

6

7

11

15

16

100

10000

1000

21

22

0

## 样例输出

10 30

6 14

7 18

11 35

15 55

16 61

100 945

10000 942820

1000 29820

21 91

22 98

# 问题分析

- 设计一个问题 **getCoins(pay, days)**, 其值表示:
  - 假设今天是第一个发 pay 个金币的日子
  - 从今天开始往后数 days 天, 这期间一共能拿到的金币数量
- 整个问题就是: **getCoins(1, Y)**

```

#include <iostream>
using namespace std;
int getCoins( int pay, int days ) { //下一次每天需发金币数, 剩余天数
    if( days <= pay ) return pay * days; //对应于连续N天
                                     //每天发金币数均为N枚
    return pay * pay + getCoins( pay + 1, days - pay);
}

int main() {
    int n;
    while(scanf("%d", &n)) {
        if (n==0) break;
        printf("%d %d\n", n, getCoins(1, n););
    }
    return 0;
}

```

# 进一步

得金币数    天数

1            1

2            2    3

3            4    5    6

4            7    8    9    10

5            11   12   13   14   15

\*\*\*\*\*  $k(k+1)/2 = n$

□ 求出的 $k$ ——最后一天的可得到的金币个数

□ 例 $k = 5$ ，只有当 $n = 15$ 时 $k = 5$ 其他小于5，所以向上取整

$$k = \frac{1}{2} \sqrt{8 * n + 1} - 1$$



# 参考程序

```
#include <stdio.h>
#include <math.h>
int main() {
    int n, i, sum, k;
    while(scanf("%d", &n) && n){
        sum = 0;
        k = ceil((-1+(double)sqrt((double)(1+8*n)))/2.0);
        for (i=1; i<=k-1; i++) {
            sum = sum+i*i;
        } //截至每天获得k-1个金币时候得到的金币总个数
        sum = sum+(n-k*(k-1)/2)*k; //每天获得k个金币的总和值
        printf("%d %d\n", n, sum);
    }
    return 0;
}
```

# POJ2694 逆波兰表达式

- **问题描述**

逆波兰表达式是一种把运算符前置的算术表达式,例如普通的表达式  $2 + 3$  的波兰表示法为  $+ 2 3$ . 波兰表达式的优点是运算符之间不必有优先级关系,也不必用括号改变运算次序,例如  $(2 + 3) * 4$  的波兰表示法为  $* + 2 3 4$ .

本题求解波兰表达式的值,其中运算符包括  $+ - * /$  四个

- **输入数据**

输入为一行,其中运算符和运算数之间都用空格分隔,运算数是浮点数

- **输出要求**

输出为一行,表达式的值

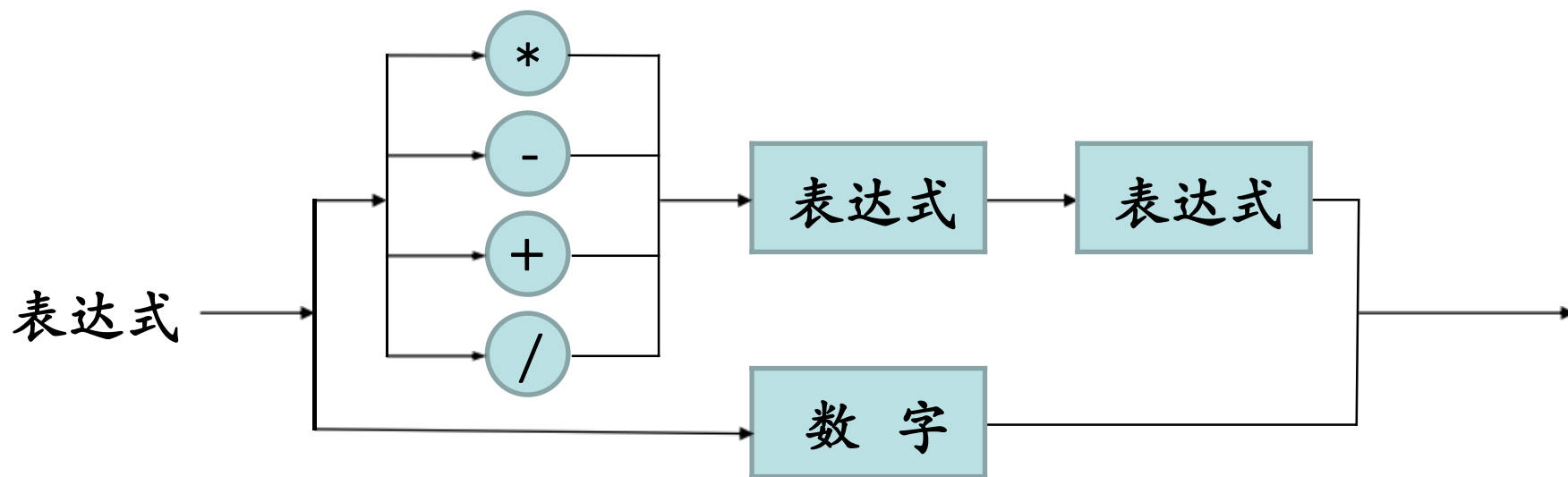
- 输入样例

\* + 11.0 12.0 + 24.0 35.0

- 输出样例

1357.000000 // (11.0 + 12.0) \* (24.0 + 35.0) = 1357.000000

## 逆波兰表达式的递归定义



```

#include <stdio>
#include<cmath>
using namespace std;

double exp() { // 读入并计算一个表达式的值
    char a[10];
    scanf("%s", a);
    switch(a[0]) {
        case '+': return exp( ) + exp( );
        case '-': return exp( ) - exp( );
        case '*': return exp( ) * exp( );
        case '/': return exp( ) / exp( );
        default: return atof(a); // 字符串转浮点数, 数字部分
    }
}

int main() {
    double ans;
    ans = exp();
    printf("%f", ans);
    return 0;
}

```

- 改写此程序, 要求将逆波兰表达式转换成常规表达式输出
- 可以包含多余的括号

```

void exp() {
    char a[10];
    scanf("%s", a);
    switch(a[0]) {
        case '+':
            printf("("); exp( ) ; printf("+"); exp(); printf(")");
            break;
        case '-':
            printf("("); exp( ) ; printf("-"); exp(); printf(")");
            break;
        case '*':
            exp( ) ; printf("*"); exp( ) ;
            break;
        case '/':
            exp( ) ; printf("/"); exp( ) ;
            break;
        default:
            printf("%s", a);
    }
}

int main() { exp(); return 0; }

```

应该给每个运算符都加括号,  
以便处理  $a-(b-c)$  和  $a/(b/c)$ ,  $a/(b*c)$  这种情况

```
void exp() {
    char a[10];
    scanf("%s", a);
    switch(a[0]) {
        case '+':
            printf("("); exp(); printf("+"); exp(); printf(")");
            break;
        case '-':
            printf("("); exp(); printf("-"); exp(); printf(")");
            break;
        case '*':
            printf("("); exp(); printf("*"); exp(); printf(")");
            break;
        case '/':
            printf("("); exp(); printf("/"); exp(); printf(")");
            break;
        default:
            printf("%s", a);
    }
}

int main() { exp(); return 0; }
```

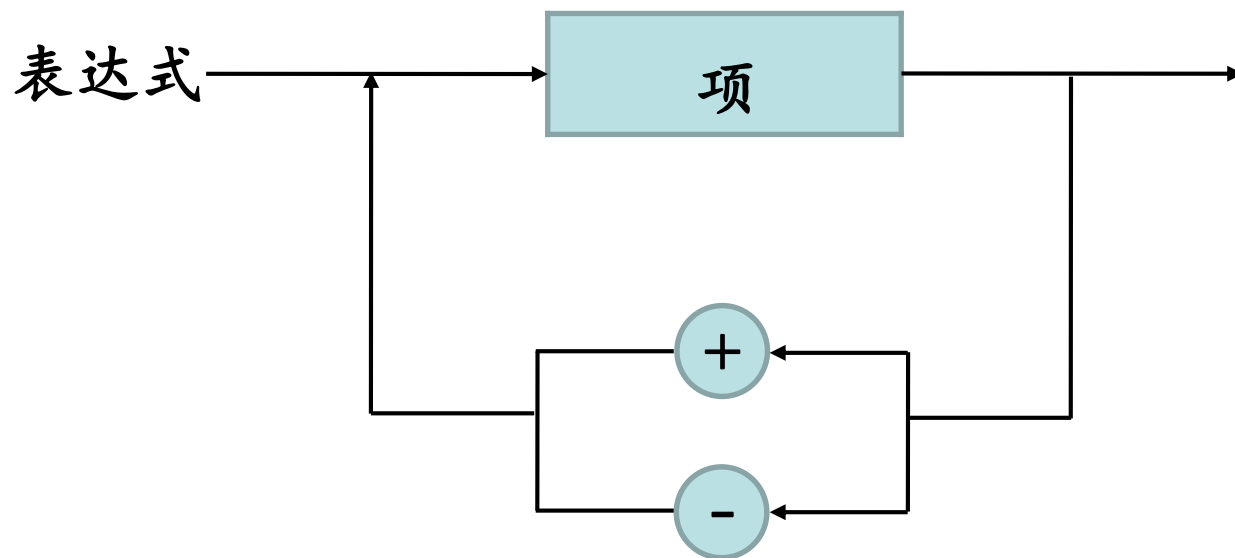
# 四则运算表达式求值

- 常规表达式计算

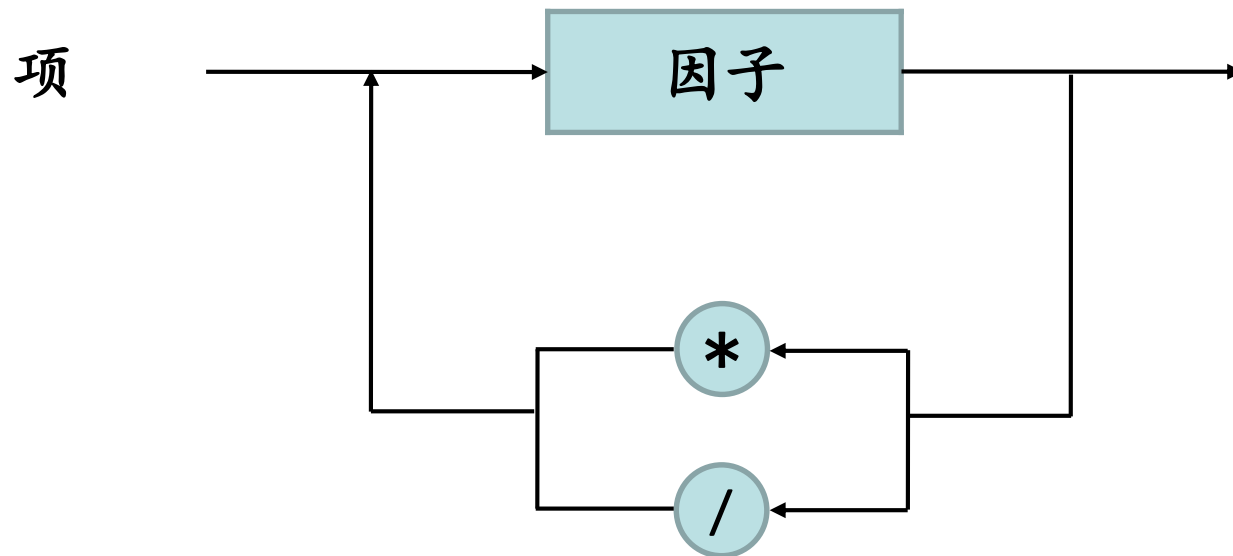
输入为普通四则运算表达式, 仅由数字, +, -, \*, /, (, ) 组成, 没有空格, 要求求其值



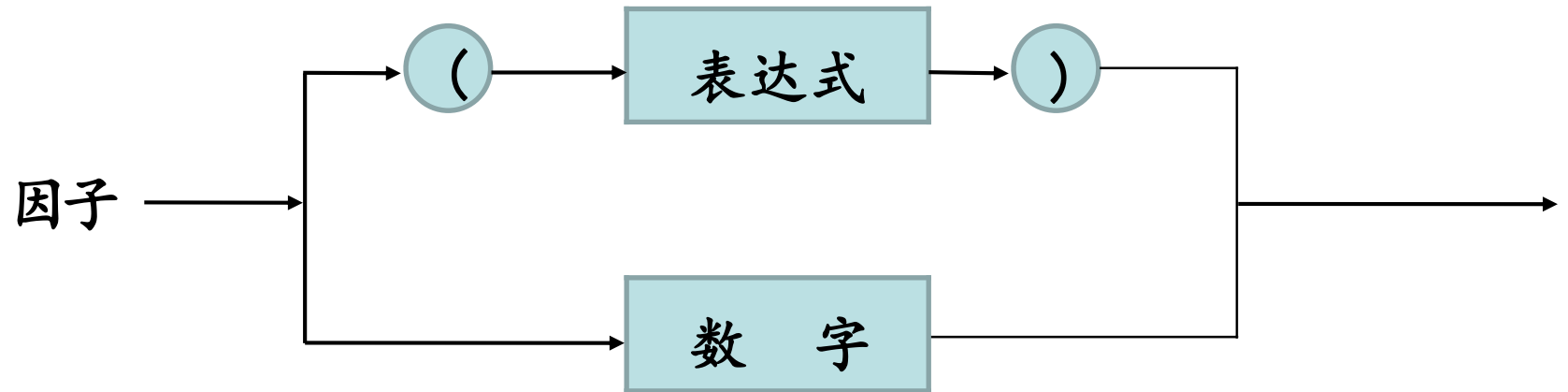
# 表达式的递归定义



# 表达式的递归定义



# 表达式的递归定义



```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cstdio>
using namespace std;

double itemValue();
double factorValue();
```

```

double expValue()
{
    //读入并计算一个表达式的值
    double v = itemValue(); //求第一项的值
    while(true) {
        char c = cin.peek(); //查看一个字符, 不从流中取走
        if( c == '-' ) {
            cin.get();
            v -= itemValue();
        }
        else if( c == '+' ) {
            cin.get();
            v += itemValue();
        }
        else
            break;
    }
    return v;
}

```

```

double itemValue ()
{ //读入并计算一个项的值
    double v = factorValue ();
    while(true) {
        char c = cin.peek();
        if( c == '*' ) {
            cin.get();
            v *= factorValue ();
        }
        else if( c == '/' ) {
            cin.get();
            v /= factorValue ();
        }
        else
            break;
    }
    return v;
}

```

```

double factorValue()
{ //读入并计算一个因子的值
    double v;
    char c = cin.peek();
    if( c == '(' ) {
        cin.get();
        v = expValue();
        cin.get();
    }
    else
        cin >> v;
    return v;
}

int main()
{
    cout << expValue() << endl;
    return 0;
}

```

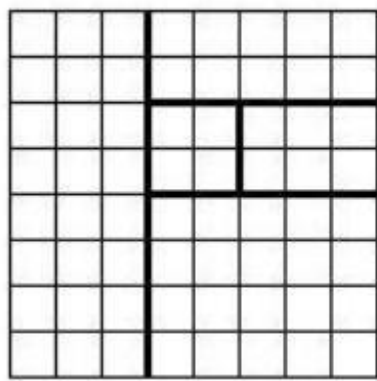
# POJ1191 棋盘分割

✂ 将一个 $8*8$ 的棋盘进行如下分割:

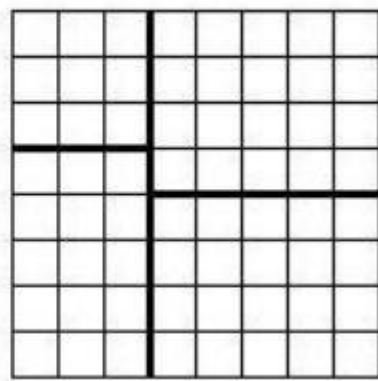
将原棋盘割下一块矩形棋盘并使剩下部分也是矩形

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



# POJ1191 棋盘分割

- ✎ 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

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

$x_i$  为第 $i$ 块矩形棋盘的总分

请编程对给出的棋盘及 $n$ ，求出 $\sigma$ 的最小值

# POJ1191 棋盘分割

## 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 $\sigma$  (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

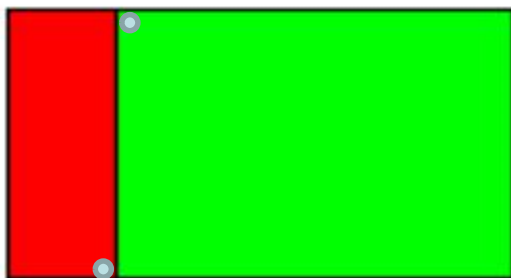
## 样例输出

1.633

# 问题分析 (1)

📁 每一次分割有以下4种方法:

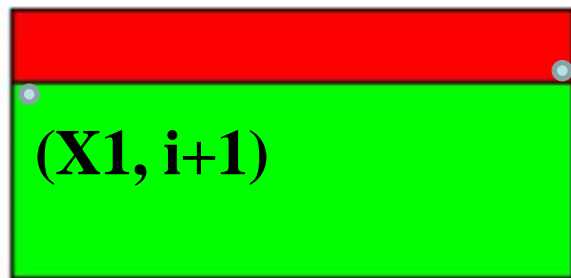
$(X1, Y1)$   $(i+1, Y1)$



$(i, Y2)$

$(X2, Y2)$

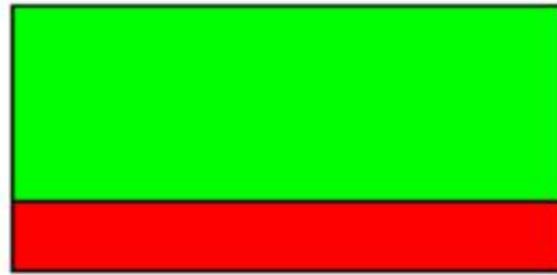
$(X1, Y1)$



$(X2, i)$

$(X1, i+1)$

$(X2, Y2)$



$f(k, \text{小棋盘}) = \{f(1, \text{割下的小棋盘}) + f(k-1, \text{待割下的棋盘})\} \quad (k \geq 2)$

k: 表示需要的份数

## 问题分析 (2)

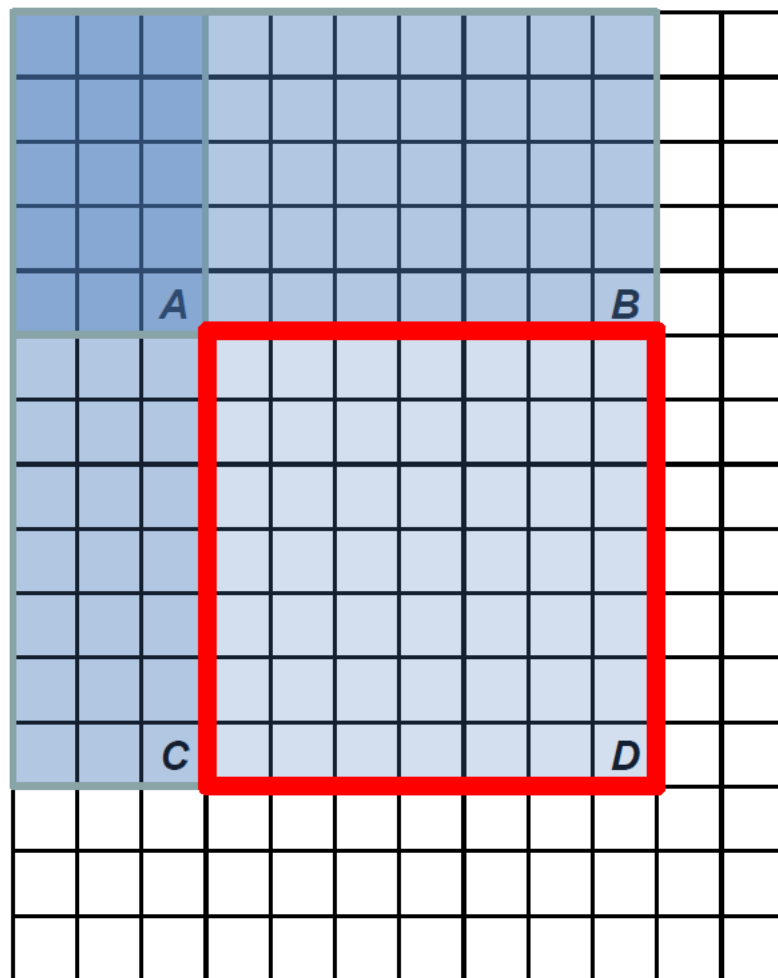
$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}}$$

如右式, 若要求出最小方差, 只需要求出最小的  $\sum x_i^2$

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

# 红色区域和的计算

$$D + A - B - C$$



# 问题分析 (3)

📖 设  $\text{fun}(n, x1, y1, x2, y2)$  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  $\text{fun}(n, x1, y1, x2, y2) =$

$$\min_{i=x1}^{x2-1} \{ \text{fun}(n-1, x1, y1, i, y2) + \text{fun}(1, i+1, y1, x2, y2) \}, \quad \text{Case 3}$$

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

$$\min_{i=y1}^{y2-1} \{ \text{fun}(n-1, x1, y1, x2, i) + \text{fun}(1, x1, i+1, x2, y2) \}, \quad \text{Case 4}$$

$$\min_{i=y1}^{y2-1} \{ \text{fun}(1, x1, y1, x2, i) + \text{fun}(n-1, x1, i+1, x2, y2) \} \quad \text{Case 2}$$

其中  $\text{fun}(1, x1, y1, x2, y2)$  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 $\text{fun}(n, x1, y1, x2, y2)$ 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 (打表)**
  - 用 $\text{res}[n][x1][y1][x2][y2]$ 来记录  $\text{fun}(n, x1, y1, x2, y2)$
  - $\text{res}$ 初始值统一为-1
  - 当需要使用 $\text{fun}(n, x1, y1, x2, y2)$ 时, 查看 $\text{res}[n][x1][y1][x2][y2]$ 
    - 如果为-1, 那么计算 $\text{fun}(n, x1, y1, x2, y2)$ , 并保存于 $\text{res}[n][x1][y1][x2][y2]$
    - 如果不为-1, 直接返回 $\text{res}[n][x1][y1][x2][y2]$

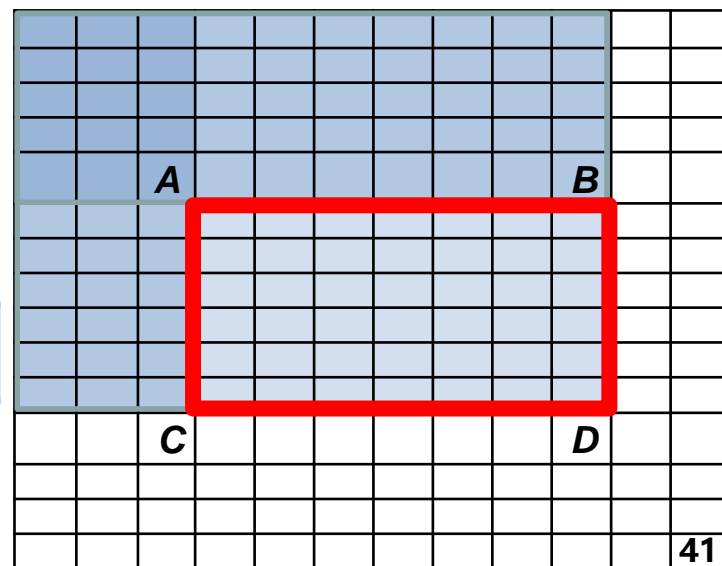


# 参考代码

```
ints[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
intres[15][9][9][9][9]; //fun的记录表

int calSum(intx1,inty1,intx2,inty2) //(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];
}
```

$$D + A - B - C$$



```
int fun(int n, int x1, int y1, int x2, int y2)
{
    int t, a, b, c, e, MIN=100000000;
    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++) { //沿x方向切
    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++) { //沿y方向切
    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;
        } //以积分图形式计算子块和, 便于calSum计算
    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;
}

```

# POJ1390 方盒游戏

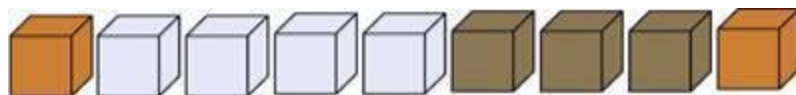


## 问题描述

N个方盒 (box) 摆成一排, 每个方盒有自己的颜色

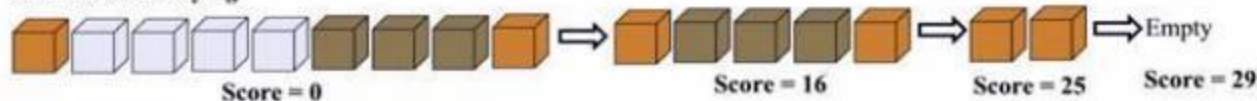
连续摆放的同颜色方盒构成一个方盒片段 (box segment)

下图中共有四个方盒片段, 每个方盒片段分别有1, 4, 3, 1个方盒

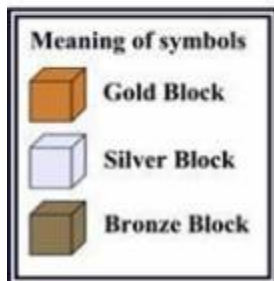
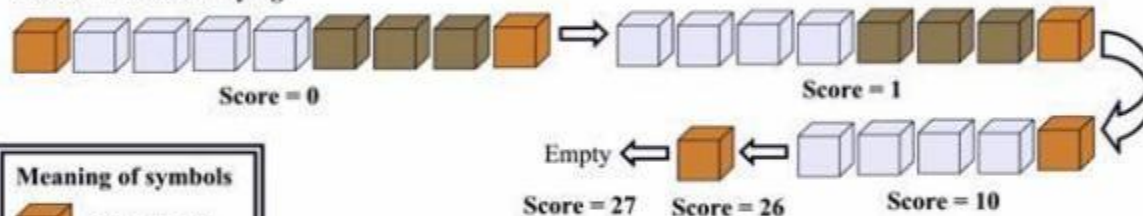


玩家每次点击一个方盒, 则该方盒所在方盒片段就会消失  
若消失的方盒片段中共有 $k$ 个方盒, 则玩家获得 $k*k$ 个积分


One Possible Playing





Another Possible Playing




 **请问:** 给定游戏开始时的状态, 玩家可获得的最高积分是多少?

 **输入:** 第一行是一个整数 $t$  ( $1 \leq t \leq 15$ ), 表示共有多少组测试数据  
每组测试数据包括两行

 第一行是一个整数 $n$  ( $1 \leq n \leq 200$ ), 表示共有多少个方盒

 第二行包括 $n$ 个整数, 表示每个方盒的颜色, 这些整数的取值范围是 $[1\ n]$

 **输出:** 对每组测试数据, 分别输出该组测试数据的序号, 以及玩家可以获得的最高积分

 **样例输入**

2

9

1 2 2 2 2 3 3 3 1

1

1

 **样例输出**

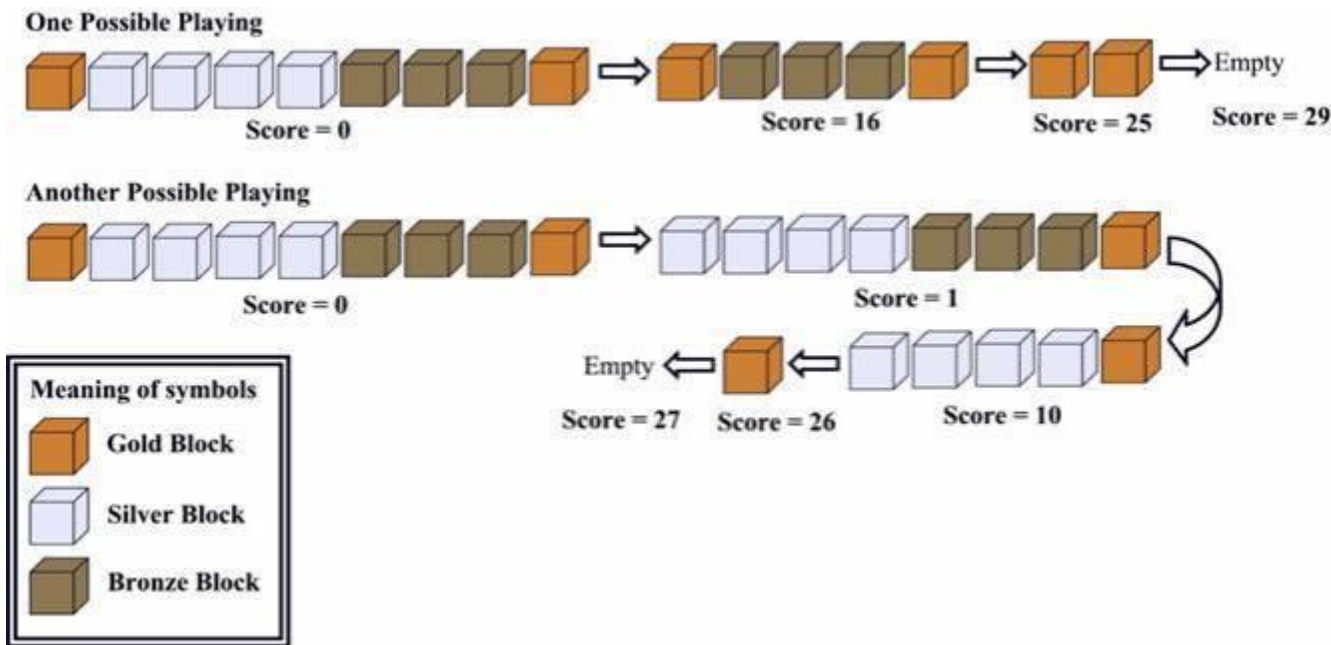
Case 1: 29

Case 2: 1

# 问题分析



当同颜色的方盒摆放在不连续的位置时，方盒的点击顺序影响玩家获得的积分



同种颜色的方盒被点击的次数越少，玩家获得的积分越高



明显的**递归问题**：每次点击之后，剩下的方盒构成一个新的方盒队列，新队列中方盒的数量减少了；然后计算玩家从新队列中可获得的最高积分

# 问题分析

点击下图中黑色方盒之前，先点击绿色方盒可提高玩家的积分：

- 同颜色方盒A和B被其他颜色的方盒隔开时，先点击其他颜色方盒，使得A和B消失前能够在同一个方盒片段中

点击下图中红色和蓝色方盒可获得的积分

- 所有红色方盒合并到同一个片段： $49+1+36=86$
- 所有蓝色方盒合并到同一个片段： $49+16+9=74$





# 问题分析

## ■ 思路:

- 将连续的若干个方块作为一个 " 大块" (box\_segment)
- 考虑, 假设开始一共有  $n$  个 " 大块", 编号0到 $n-1$ 
  - 第 $i$ 个大块的颜色是  $\text{color}[i]$
  - 包含的方块数目, 即长度, 是 $\text{len}[i]$
- $\text{click\_box}(i, j)$ 表示从大块 $i$ 到大块 $j$ 这一段消除后所能得到的最高分
- 则整个问题就是:  **$\text{click\_box}(0, n-1)$**

# 问题分析

要求click\_box(i, j)时, 考虑最右边的大块j,

对它有两种处理方式, 要取其优者:

1) 直接消除它, 此时能得到最高分就是:

$$\text{click\_box}(i, j-1) + \text{len}[j]*\text{len}[j]$$

2) 期待以后它能和左边的某个同色大块合并

# 问题分析

- 考虑和左边的某个同色大块合并:
  - 左边的同色大块可能有很多个, 到底和哪个合并最好? 不知道, 只能枚举
  - 假设 大块 $j$  和 左边的大块 $k$  ( $i \leq k < j-1$ ) 合并, 此时能得到的最高分是多少呢?

# 问题分析

- 考虑和左边的某个同色大块合并:
  - 左边的同色大块可能有很多个, 到底和哪个合并最好? 不知道, 只能枚举
  - 假设 大块j 和 左边的大块k ( $i \leq k < j-1$ ) 合并, 此时能得到的最高分是多少呢?

# 问题分析

- 考虑和左边的某个同色大块合并:
  - 左边的同色大块可能有很多个, 到底和哪个合并最好? 不知道, 只能枚举
  - 假设 **大块j** 和 **左边的大块k ( $i \leq k < j-1$ )** 合并, 此时能得到的最高分是多少呢?

是不是:

$$\text{click\_box}(i, k-1) + \text{click\_box}(k+1, j-1) + (\text{len}[k] + \text{len}[j])^2$$

# 问题分析

$$\text{click\_box}(i, k-1) + \text{click\_box}(k+1, j-1) + (\text{len}[k] + \text{len}[j])^2$$

- 不对!
- 因为将大块 k 和大块 j 合并后, 形成新大块会在最右边  
将该新大块直接将其消去的做法, 才符合上述式子
- 但直接将其消去, 未必是最好的, 也许它还应该和左边的同色大块合并, 才更好
- 递推关系无法形成, 怎么办?

# 问题分析

□ 需要改变问题的形式:

**click\_box(i, j)** 这个形式不可取, 因为无法形成递推关系

□ 考虑新的形式:

**click\_box(i, j, ex\_len)**

表示:

大块j的右边已经有一个长度为**ex\_len**的大块

(该大块可能是在合并过程中形成的, 不妨就称其为**ex\_len**),

**且j的颜色和ex\_len相同**, 在此情况下将i 到j 以及**ex\_len**都消除所能得到的最高分

□ 于是整个问题就是求: **click\_box(0, n-1, 0)**

# 问题分析

□ 求 `click_box(i, j, ex_len)` 时, 有两种处理方法,  
取最优者假设 大块j 和 `ex_len` 合并后的大块称作 Q

1) 将Q直接消除, 这种做法能得到的最高分就是:

$$\text{click\_box}(i, j-1, 0) + (\text{len}[j] + \text{ex\_len})^2$$

2) 期待Q以后能和左边的某个同色大块合并  
需要枚举可能和Q合并的大块

假设让大块k和Q合并, 则此时能得到的最大分数是:

$$\text{click\_box}(i, k, \text{len}[j] + \text{ex\_len}) + \text{click\_box}(k+1, j-1, 0)$$

□ 递归的终止条件是什么?



# 问题分析

- `click_box(i, j, ex_len)` 递归的终止条件：  
 $i == j$

# 参考程序1

```
#include<cstring>
#include<iostream>
using namespace std;
struct box_segment {
    int color;

    int len;
};

struct box_segment segment[200];
```

```
int click_box(int start, int end, int extra_len) {  
    int i;  
    int result, temp;  
    result = segment[end].len + extra_len;  
    result = result*result; //end和extra_len一起消去的得分  
    if ( start == end ) return result;  
  
    result += click_box(start, end-1, 0);  
    //直接消除 end 和 extra_len的合并块
```

```
for ( i = end - 1; i >= start; i-- ) {  
    if ( segment[i].color!=segment[end].color ) continue;  
    temp = click_box(start,i, segment[end].len + extra_len)  
        + click_box(i+1, end-1, 0);  
    if ( temp<=result ) continue;  
    result = temp;  
    break;  
}  
return result;  
}
```

```

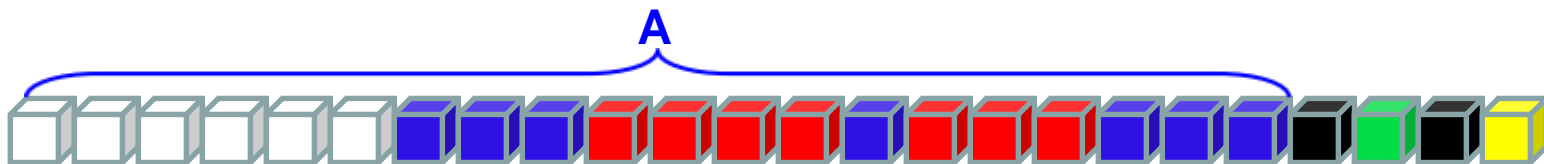
int main(int argc, char *argv[]){
    int t, n, i, j, end, color;
    cin >> t;
    for ( i=0; i<t; i++ ) {
        cin >> n; end = 0; cin >> segment[end].color;
        segment[end].len = 1;
        for ( j=1; j<n; j++ ) {
            cin >> color;
            if ( color==segment[end].color ) segment[end].len++;
            else {
                end++;
                segment[end].color = color;
                segment[end].len = 1;
            }
        }
        cout << "Case " << i+1 << ": " << click_box(0, end, 0)
            << endl;
    }
    return 0;
}

```

# 避免递归计算中的重复计算

- 📁 在POJ上, 上述程序会**超时**
- 📁 考虑下图表示的测试数据: 根据上述程序, 将2次计算A部分可获得最高积分
  - 先点击右边的黑色方盒后, 依次点击绿色和黑色方盒, 然后计算A部分可获得的最高积分
  - 先点击绿色方盒, 再点击黑色方盒, 计算A部分可获得的最高积分
- 📁 为此, 应将递归计算的结果保留下来, 避免重复的计算
- 📁 递归函数包括三个参数 **start, end, extra\_len**, 则用一个**三维数组**存储递归的中间结果, 每一维的大小为对应参数的取值范围

**int click\_box(int start, int end, int extra\_len)**



# 参考程序2

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

struct box_segment {
    int color;

    int len;
};

struct box_segment segment[200];
int score[200][200][200];
```

```

int click_box(int start, int end, int extra_len) {
    int i, result, temp;
    if ( score[start][end][extra_len]>0 ) return score[start][end][extra_len];
    result = segment[end].len + extra_len;
    result = result*result;
    if (start==end) { score[start][end][extra_len]= result; return
        score[start][end][extra_len]; }
    result += click_box(start, end-1, 0);
    for ( i = end - 1; i >= start; i-- ) {
        if (segment[i].color!=segment[end].color) continue;
        temp = click_box(start, i, segment[end].len + extra_len)+click_box(i+1, end-1, 0);
        if ( temp<=result ) continue;
        result = temp;
        break;
    }
    score[start][end][extra_len] = result;
    return score[start][end][extra_len];
}

```



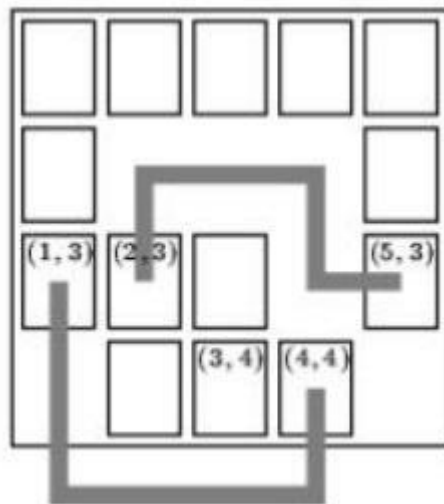
```

int main(int argc, char *argv[]){
    int t, n, i, j, end, color;
    cin >> t;
    for (i=0; i<t; i++) {
        cin >> n; end = 0; cin >> segment[end].color; segment[end].len = 1;
        for (j=1; j<n; j++) {
            cin >> color;
            if ( color==segment[end].color ) segment[end].len++;
            else {
                end++; segment[end].color = color; segment[end].len = 1;
            }
        }
        memset(score, 0, sizeof(score));
        cout << "Case " << i+1 << ": " << click_box(0, end, 0) << endl;
    }
    return 0;
}

```

## ■ 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
- 如图所示,每个正方格子上可以有一张游戏卡片,当然也可以没有
- 当下面的情况满足时,我们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



你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

- 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 $h$  ( $1 \leq w, h \leq 75$ ), 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

- 之后每行上包括4个整数 $x1, y1, x2, y2$  ( $1 \leq x1, x2 \leq w, 1 \leq y1, y2 \leq h$ )。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
- 然后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 "ksegments.", 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

XXXXX

X X

XXX X

XXX

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.

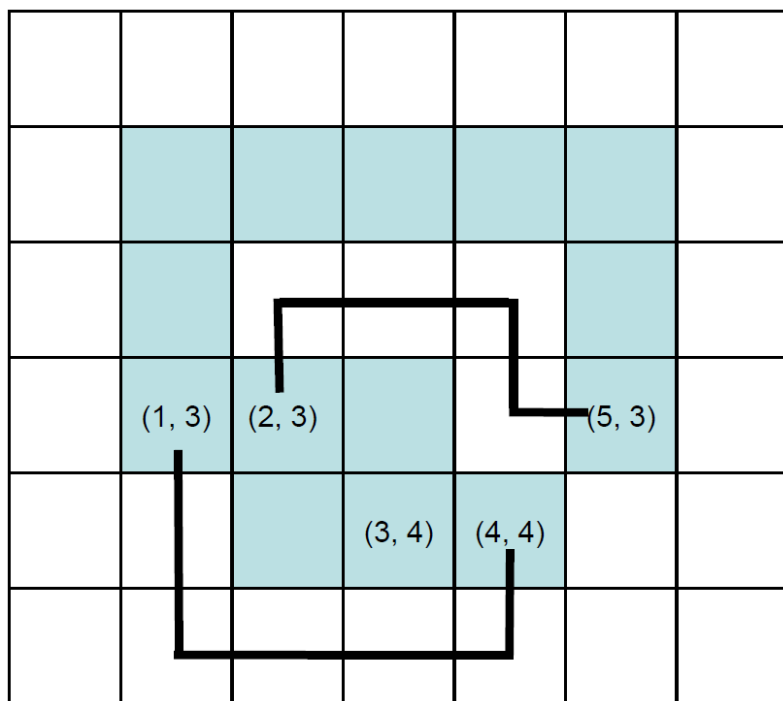
Pair 2: 3 segments.

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**穷举方式**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
- 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)

	X	X	X	X	X	
	X				X	
	X	X			X	
		X	X	X		

# 问题分析 (5)

📁 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 ++){
```

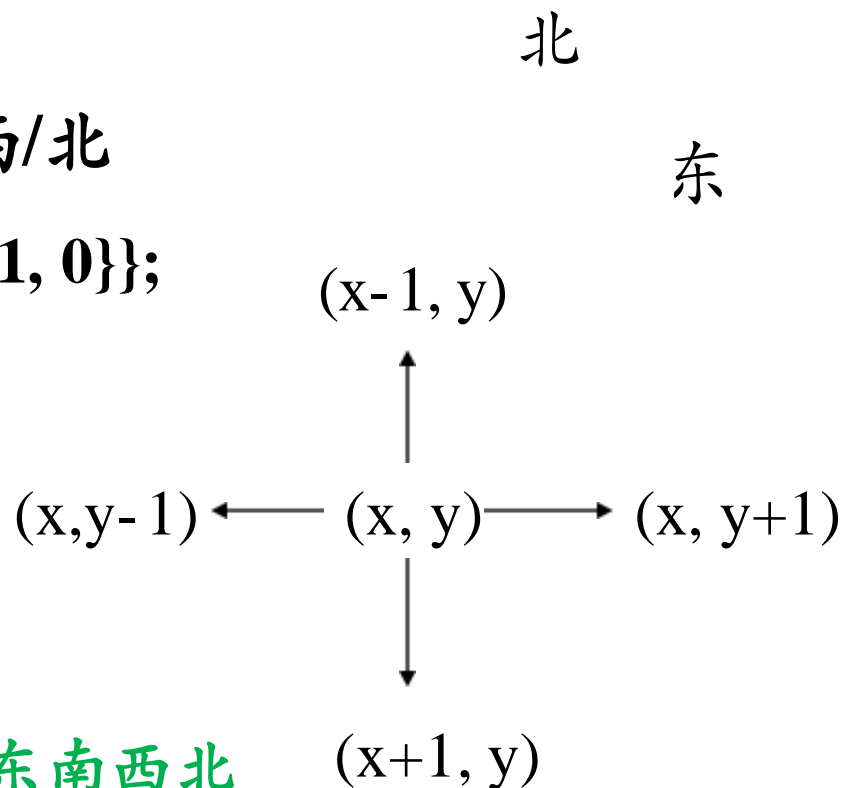
```
    int x = now_x + to[i][0];
```

```
    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 (6)

## 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

$((\text{board}[y][x] == ' ') \ \&\& \ (\text{mark}[y][x] == \text{false}))$

**T3:** 已经到达终点

$(x == \text{end\_x}) \ \&\& \ (y == \text{end\_y}) \ \&\& \ (\text{board}[y][x] == 'X')$

综上, (x, y)有效的条件是 **T1 && (T2 || T3)**

$((x > -1) \ \&\& \ (x < w + 2) \ \&\& \ (y > -1) \ \&\& \ (y < h + 2)$

$\ \&\& \ (((\text{board}[y][x] == ' ') \ \&\& \ (\text{mark}[y][x] == \text{false})))$

$\ || \ ((x == \text{end\_x}) \ \&\& \ (y == \text{end\_y}) \ \&\& \ (\text{board}[y][x] == 'X'))))$

# 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')))) { //如果新位置有效
        mark[y][x] = true; //标记该位置已经经过
                            //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);
        inti, 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;
}

```

# 放苹果的递归解法

把  $M$  个同样的苹果放在  $N$  个同样的盘子里, 允许有的盘子空着不放, 问共有多少种不同的分法?

(用 $K$ 表示) 5, 1, 1和1, 5, 1 是同一种分法

```
int AppleWays(int m, int n)
{
    // m apples, n plates
    if( m == 0 )
        return 1;
    if( n == 0 )
        return 0;
    if( n > m )
        return AppleWays(m, m);
    int tmp = AppleWays(m, n-1);
    return tmp + AppleWays(m-n, n);
}
```

# 放苹果问题用栈模拟递归的解法

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 (1),(2),(3) 就是返回地址

```
int AppleWays(int m,int n)
{
    //m apples, n plates
    if( m == 0 ) return 1;
    if( n == 0 ) return 0;
    if( n > m ) return AppleWays(m,m); // (1)
    int tmp = AppleWays(m,n-1); // (2)
    return tmp + AppleWays(m-n,n); // (3)
}
```

# 用栈模拟放苹果的递归解法

```
struct Node { //栈中要放的东西
    int m,n;
    int tmp; //局部变量tmp
    int retAdr; //返回地址,为0则说明还没开始算
    Node() { }
    Node(int m_,int n_,int adr):m(m_),n(n_),retAdr(adr){
    }
};
```

# 用栈模拟放苹果的递归解法

```
int StkWays(int m,int n)
{
    Node nd;
    int retVal; //某层 StkWays(i,j)的最终结果
    stack<Node> stk;
    stk.push(Node(m,n,0)); //要计算 StkWays(m,n)
    while(!stk.empty()) {
        nd = stk.top();
        if( nd.m == 0) {
            retVal = 1;
            stk.pop();
        }
        else if( nd.n == 0) {
            retVal = 0;
            stk.pop();
        }
    }
}
```

```
int AppleWays(int m,int n)
{
    //m apples, n plates
    if( m == 0) return 1;
    if( n == 0) return 0;
    if(n>m) return AppleWays(m,m); //(1)
    int tmp = AppleWays(m,n-1); //(2)
    return tmp + AppleWays(m-n,n); //(3)
}
```

# 用栈模拟放苹果的递归解法

```
else {
```

```
    if( nd.retAdr == 0 ) { //还没开始算
```

```
        if( nd.n > nd.m) {
```

```
            stk.top().retAdr = 1;
```

```
            stk.push(Node(nd.m,nd.m,0));
```

```
        }
```

```
    else {
```

```
        stk.top().retAdr = 2;
```

```
        stk.push(Node(nd.m,nd.n-1,0));
```

```
    }
```

```
}
```

```
else if( nd.retAdr == 1) stk.pop();
```

```
else if( nd.retAdr == 2) {
```

```
    stk.top().tmp = retVal;
```

```
    stk.top().retAdr = 3;
```

```
    stk.push(Node(nd.m-nd.n,nd.n,0));
```

```
}
```

```
int AppleWays(int m,int n)
{
    //m apples, n plates
    if( m == 0) return 1;
    if( n == 0) return 0;
    if(n>m) return AppleWays(m,m); //(1)
    int tmp = AppleWays(m,n-1); //(2)
    return tmp + AppleWays(m-n,n); //(3)
}
```

# 用栈模拟放苹果的递归解法

```
        else if(nd.retAdr == 3) {
            retVal += nd.tmp;
            stk.pop();
        }
    }
}
return retVal;
}
```

```
int AppleWays(int m,int n)
{
    //m apples, n plates
    if( m == 0) return 1;
    if( n == 0) return 0;
    if(n>m) return AppleWays(m,m); //(1)
    int tmp = AppleWays(m,n-1); //(2)
    return tmp + AppleWays(m-n,n); //(3)
}
```

## ■ 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



## ■ 递归的优缺点

- 优点: 直接, 算法程序结构清晰, 思路明了
- 缺点: 递归函数的每一次调用都要把分配的相应空间保存起来

## ■ 递归运行过程大致如下:

- 1) 计算当前函数的实参的值
- 2) 分配空间, 并将首地址压栈, 保护现场
- 3) 转到函数体, 执行各语句, 此前部分会重复发生 (递归调用)
- 4) 直到出口, 从栈顶取出相应数据, 包括 返回地址, 返回值等, 收回空间, 恢复现场, 转到上一层的调用位置继续执行本次调用未完成的语句

**Thanks !**

