



程序设计实习

郭炜 微博 <http://weibo.com/guoweiofpku>
<http://blog.sina.com.cn/u/3266490431>



递归

用递归替代循环

输出整数1-N的全排列

```
void Permutation(int k) {
    if( k == N) {
        for( int i = 0;i < N; ++ i )
            cout << result[i] << " " ;
        cout << endl;
        return ;
    }
    for( int i = 1;i <= N; ++ i ) {
        if( !used[i]) {
            result[k] = i;
            used[i] = 1;
            Permutation(k+1);
            used[i] = 0;
        }
    }
}
```

```
#include <iostream>
using namespace std;
int result[200];
int N;
int used[200] = {0};
int main()
{
    cin >> N;
    Permutation(0);
}
```

求全排列的 $n!$ 的写法

求 $S = \{a_1, a_2, a_3 \dots a_n\}$ 的全排列，等于：

```
for (int i = 1; i <= n; ++i ) {  
    for x in (S -  $a_i$ ) 的每个排列  
        cout <<  $a_i$  + x;  
}
```

求全排列的 $n!$ 的写法

```
#include <iostream>
#include <algorithm>
using namespace std;
template <class T>
void Perm(T s, T e, T k)
{ //依次生成区间[k,e)的全排列
    if( k == e ) {
        for(s; s != e; ++s)
            cout << *s << " ";
        cout << endl;
        return;
    }
    for(T i = k ; i != e; ++i ) {
        swap( *k, *i);
        Perm(s,e,k+1);
        swap(*k,*i);
    }
}
```

求全排列的 $n!$ 的写法

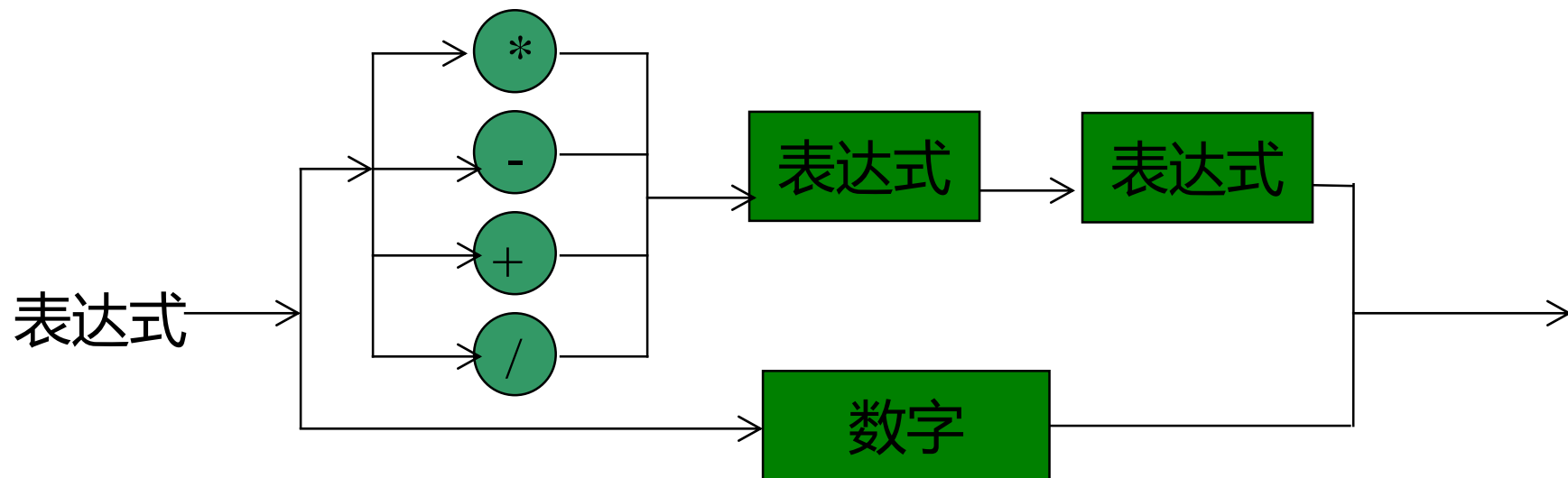
```
int main()
{
    char a[] = "abcd";
    Perm(a,a+4,a);
    return 0;
}
```

解决以递归形式定义的问题

例题： openjudge [2694](#) 波兰表达式

- 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

波兰表达式的递归定义



* + 2 3 4

```
#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
    }
}
```

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

- 改写此程序，要求将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; }
```

```
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);
    }
}

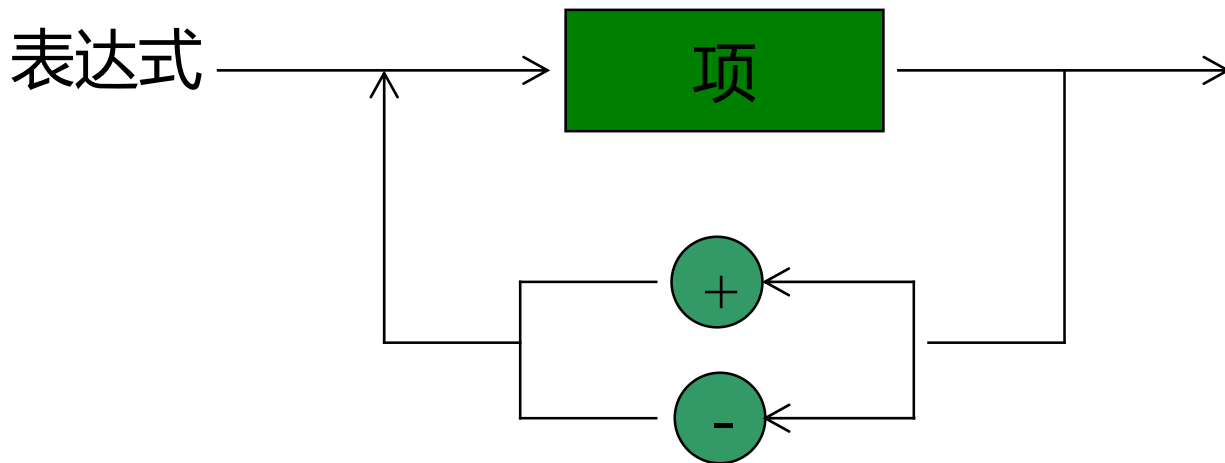
int main() { exp(); return 0; }
```

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

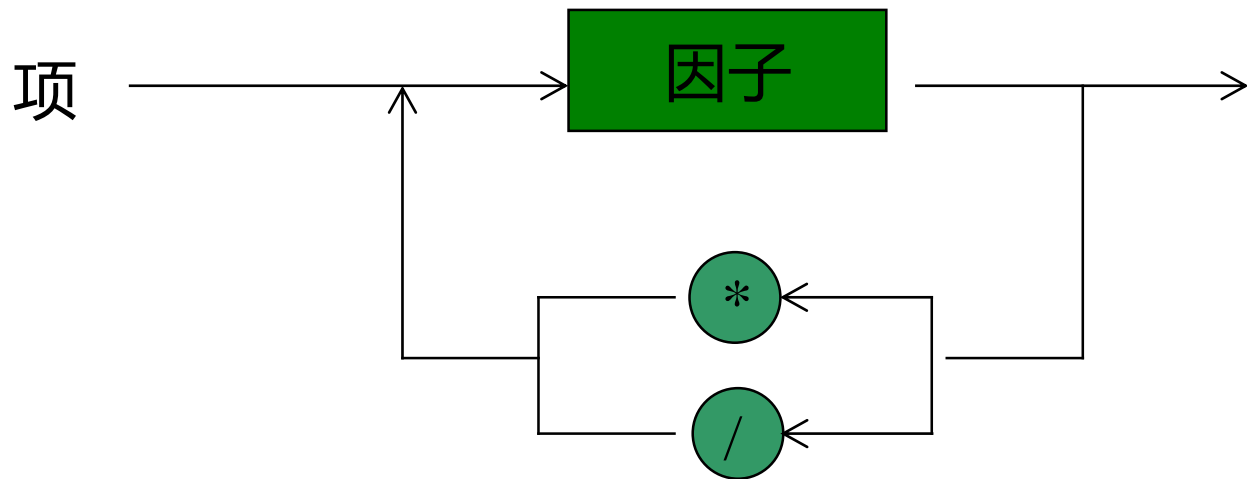
常规表达式计算

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

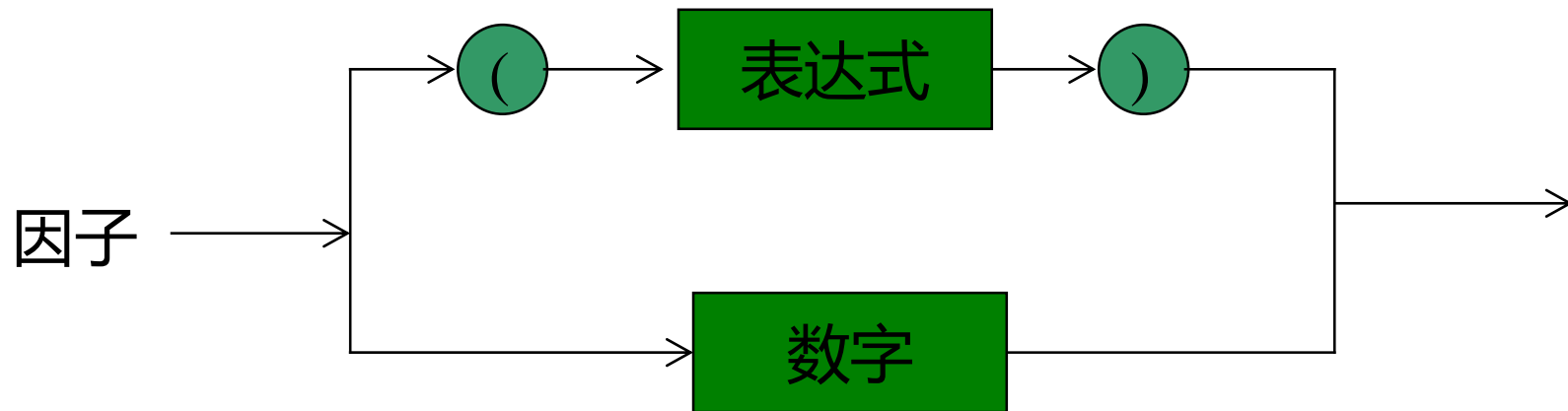
表达式的递归定义



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;  
}
```

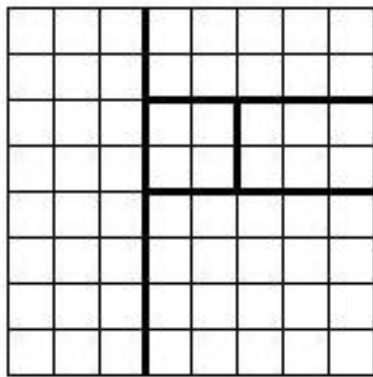
```
double factorValue()  
{ //读入并计算一个因子的值  
    double v;  
    char c = cin.peek();  
    if( c == '(' ) {  
        cin.get();  
        v = expValue();  
        cin.get();  
    }  
    else  
        cin >> v;  
    return v;  
}  
  
int main()  
{  
    cout << expValue() <<endl;  
    return 0;  
}
```



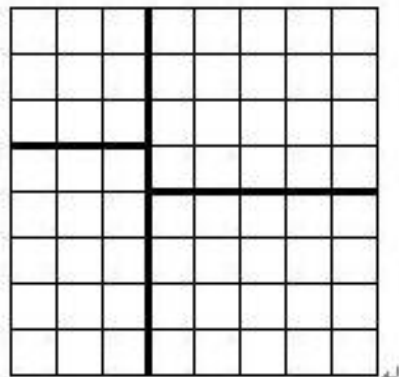
用递归进行穷举

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小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小棋盘共有 n 块矩形小棋盘。（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，一块矩形小棋盘的总分为其所含各格分值之和。现在需要把原棋盘按上述规则分割成 n 块矩形小棋盘，并使各矩形小棋盘总分的均方差最小。

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

x_i 为第 i 块矩形小棋盘的总分。
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

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

如右式，若要求出
最小方差，只需要
求出最小的 $\sum x_i^2$

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

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

问题分析

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

其中 $\text{fun}(1, x1, y1, x2, y2)$ 等于该棋盘内分数和的平方

且当 $x2-x1+y2-y1 < n-1$ 时, $\text{fun}(n, x1, y1, x2, y2) = +\infty$

问题分析

当 $x_2 - x_1 + y_2 - y_1 < n - 1$ 时, $\text{fun}(n, x_1, y_1, x_2, y_2) = +\infty$

要分成 n 块, 即要切 $n-1$ 刀,

X方向最多可切 $x_2 - x_1$ 刀, y方向最多可切 $y_2 - y_1$ 刀

则必须 $x_2 - x_1 + y_2 - y_1 \geq n - 1$

那么 $x_2 - x_1 + y_2 - y_1 < n - 1$ 时即无法再分

问题分析

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

```
#include <iostream>
#include <cmath>
#include <cstring>
#include <iomanip>
using namespace std;
int sum[9][9]; //sum[i][j]是从(1,1)到 (i,j)的总分
int result[16][9][9][9][9];
int smallSum(int x1,int y1,int x2,int y2)
{ //求(x1,y1) 到(x2,y2)的总分
    return sum[x2][y2] - sum[x1-1][y2] - sum[x2][y1-1] +
           sum[x1-1][y1-1];
}
```



```
int fun(int n,int x1,int y1,int x2,int y2)
{ //求将 (x1,y1)-(x2,y2)分割成 n块,
  //所能的到的n个块的分数的平方之和的最小值
    int v = 1 << 30;
    if( n == 1) {
        int t = smallSum(x1,y1,x2,y2);
        return t * t;
    }
    if( x2-x1+y2-y1 < n-1)
        return v;
    if( result[n][x1][y1][x2][y2] != -1)
        return result[n][x1][y1][x2][y2];
```

```
for( int x = x1; x < x2; ++x) {
    int left = smallSum(x1,y1,x,y2);
    int right = smallSum(x+1,y1,x2,y2);
    int t = min(fun(n-1,x+1,y1,x2,y2) + left*left,
                fun(n-1,x1,y1,x,y2) + right*right);
    v = min(t,v);
}
for(int y = y1; y < y2; ++y) {
    int up = smallSum(x1,y1,x2,y);
    int down = smallSum(x1,y+1,x2,y2);
    int t = min(fun(n-1,x1,y+1,x2,y2) + up*up,
                fun(n-1,x1,y1,x2,y) + down*down);
    v = min(t,v);
}
result[n][x1][y1][x2][y2] = v;
return v;
}
```

```
int main()
{
    int n;
    memset(sum,0,sizeof(sum));
    memset(result,0xff,sizeof(result));
    cin >> n;
    for(int i = 1;i <= 8; ++i) {
        int rowSum = 0;
        for(int j = 1; j <= 8; ++j) {
            int s;
            cin >> s;
            rowSum += s;
            sum[i][j] = sum[i-1][j] + rowSum;
        }
    }
}
```

```

int squareSum = fun(n,1,1,8,8);
double result = n * squareSum - sum[8][8] * sum[8][8];
cout << setiosflags(ios::fixed) << setprecision(3) <<
      sqrt(result/(n*n));
}

```

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

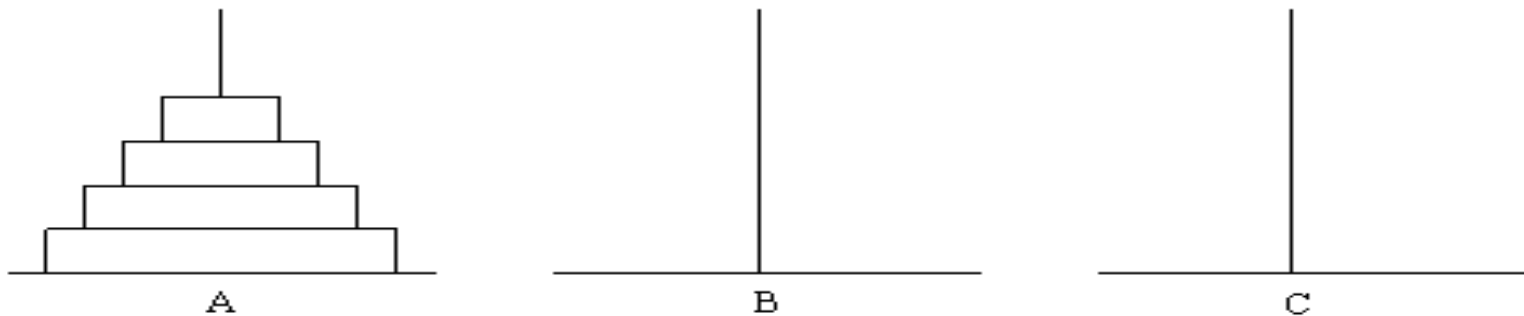
$$\begin{aligned}
& \sum (x_i - \bar{x})^2 \\
&= \sum (x_i^2 - 2x_i\bar{x} + \bar{x}^2) \\
&= \sum x_i^2 - \sum 2x_i\bar{x} + n\bar{x}^2 \\
&= \sum x_i^2 - 2n\bar{x}^2 + n\bar{x}^2 \\
&= \sum x_i^2 - n\bar{x}^2
\end{aligned}$$



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

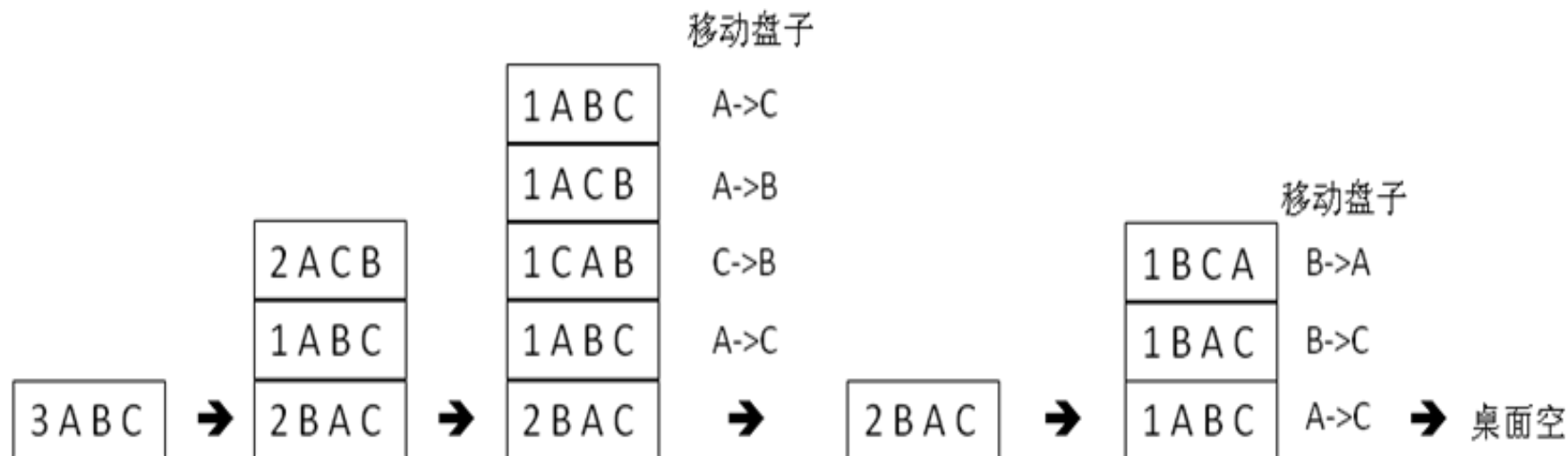
```
#include <iostream>
using namespace std;
void Hanoi(int n, char src, char mid, char dest)
//将src座上的n个盘子，以mid座为中转，移动到dest座
{
    if( n == 1) { //只需移动一个盘子
        cout << src << "->" << dest << endl; //直接将盘子从src移动到dest即可
        return ; //递归终止
    }
    Hanoi(n-1, src, dest, mid); //先将n-1个盘子从src移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

```

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) ;
}
```

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 Ways(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)  
}
```