递归 1

递归是算法的基础。能够灵活运用好递归，意味着绝大部分算法可以较为容易地掌握，了解并熟练递归的思维方式，可以对接到动态规划的解题思路。毋庸置疑，递归是非常重要的一节内容，是从语言入门到算法入门的一道门槛。

递归的概念

一个函数、过程、概念或数据结构，如果在其定义或说明内部直接或间接地出现有其本身的引用，或者是为了描述问题的某一状态，必须用到它的上一状态，而描述上一状态，又必须用到它的上一状态……这种用自己来定义的方法，称之为递归或者递归定义。

比如：阶乘的定义为

$$n! = \begin{cases} (n-1)! * n, & n > 0 \\ 1, & n = 0 \end{cases}$$

在这个定义中，为了描述 n 的阶乘，用到了 n-1 的阶乘，进一步，为了描述 n-1 的阶乘，则会用到 n-2 的阶乘。

再比如：斐波拉契数列的定义：

$$f(n) = \begin{cases} f(n-1) + f(n-2), & n > 2 \\ 1, & n = 1 \text{ 或者 } n = 2 \end{cases}$$

在这个定义中，f(n)用了 f(n-1)+f(n-2)来定义。

在程序设计中，过程或函数直接或者间接调用自己，就称为递归调用

递归的过程

递归过程实际上借助于一个递归工作栈来实现的。

首先问题向一个方向一步一步分解，问题规模逐渐降低，这样，问题向一极推进的过程称之为递归过程；

然后这些问题又按后提出先解决的顺序依次得到解决，最后得到原问题的解。这样，在子问题得到逐一解决的基础上，最后回到原问题的解的变化过程，称之为回归过程。

递归定义的要素

一个递归定义，必须包含两大要素。

1、递归边界或终止条件。如果没有边界，递归将无限循环下去，没完没了。

2、递归定义，使问题向边界条件转化的规则。

例题 1：用递归的方式求 n 的阶乘。输入 n，输出 n 的阶乘

代码：

```
#include<iostream>
using namespace std;
int n;
int jc(int n){
    if(n==0)return 1;  //边界条件
    return jc(n-1)*n;  //转换规则,n 的阶乘等于 n-1 的阶乘乘以 n
}
int main(){
    cin>>n;
    cout<<jc(n);
}
```

例题 2：输入 n，输出斐波那契数列的第 n 项。

代码：

```
#include<iostream>
using namespace std;
int f(int n){
    if(n==1||n==2)return 1;  //边界条件
    return f(n-1)+f(n-2);  //转换规则
}
int main(){
    int n;
    cin>>n;
    cout<<f(n);
}
```

例题 3：用递归方式求两个数的最大公约数

分析：

用递归的方式求两个数的最大公约数，可能不知如何定义递归规则。如果细推辗转相除法求最大公约数，知道 a 和 b 的最大公约数等价于 b 和余数 r 的最大公约数，则可以得出递归函数：

$$gcd(a,b)=\begin{cases} b, & a\%b==0 \\ gcd(b,a\%b), & a\%b!=0 \end{cases}$$

根据这个函数实现代码即可。

代码：

```
#include<iostream>
using namespace std;
int gcd(int a,int b){
    if(a%b==0)return b;  //边界条件
    return gcd(b,a%b);  //转换规则
}
int main(){
    int a,b;
    cin>>a>>b;
    cout<<gcd(a,b);
}
```

深入理解递归调用

我们可能会不明白一个函数调用自己，是怎么实现的。函数每调用一次自己，实际上是新开辟了一块新的内存空间运行新调用的程序，再调用，再开辟，看似一样的代码，实际是在不同的内存空间运行的。打个比方。有一栋楼，每层楼里面的所有装修都是一模一样的，每层楼之间都只有一个接口。我们现在使用 jc(5)这个函数，相当于在这栋楼的 5 楼，当 jc(5)调用 jc(4)时，相当于从 5 楼下到了 4 楼，虽然 4 楼的装修跟 5 楼是一模一样的，但是毕竟是两个不同的空间，如果你在四楼打坏了一个花瓶，是不会影响到 5 楼的花瓶的，意味着你修改 4 楼的局部变量的值，是不会影响到 5 楼的局部变量的值。

为了弄清楚这个问题，我们结合以下例题讲解。

例题：用递归方式实现输入一串字符串（以符号’@’表示字符串结束），输出它的反序

代码：

```
#include<iostream>
using namespace std;
void work(){
    char ch;
```

```
5.      cin>>ch;
6.      if(ch=='@')return;
7.      work();
8.      cout<<ch;
9.  }
10. int main(){
11.     work();
12. }
```

我们以输入“ABC@”为例来讲解以上代码。首先程序运行，即调用 work()函数。

1、第一次进入 work()函数，我们理解为来到了 1 楼，在这一楼里，声明了一个变量 ch，然后读入一个变量到 ch，因为我们是按“ABC@”来输入，首先读到的是“A”，这是 ch 就是“A”。然后运行代码第 6 行，判断 ch 是否为“@”，在这里不为“@”，继续调用 work()函数。注意，这一层的 work 函数还没有运行结束，余下第 8 行 cout<<ch;还没有执行的。

2、因为调用了 work()函数，相当于我们从 1 楼来到了 2 楼，在这一层里的代码跟第一层的代码是一模一样的，但是空间是不一样的。在这一层里，也声明了一个变量 ch，也读入一个字母“B”到变量 ch 中。这一层 ch 的值为“B”，这个 ch 跟 1 楼的 ch 是两个不同的变量，尽管他们的名字相同。接着执行判断，因为 ch 的值不为‘@’，继续调用 work()函数。注意，这一层的第 8 行语句也还没执行的。

3、还是调用 work()函数，仿佛从 2 楼上到了 3 楼，还是一模一样的装修，在 3 楼，还是有一个叫 ch 的变量，但是我们知道，这个 ch 跟 2 楼的 ch，跟 1 楼的 ch 不是同一个变量。同样通过读入给 ch 赋值为‘C’。判断 ch 是否为‘@’，继续调用 work()函数。注意，这一层的第 8 行语句也还没执行的。

4、通过上面的调用，我们现在上到了 4 楼，在 4 楼继续有一个变量 ch，读入字符‘@’，赋值给 ch，然后判读 ch 是否为‘@’，在这里刚好相等了，执行了 return 语句。一旦执行 return 语句，我们相当于从 4 楼走回了 3 楼。4 楼剩下的语句也就不执行了。

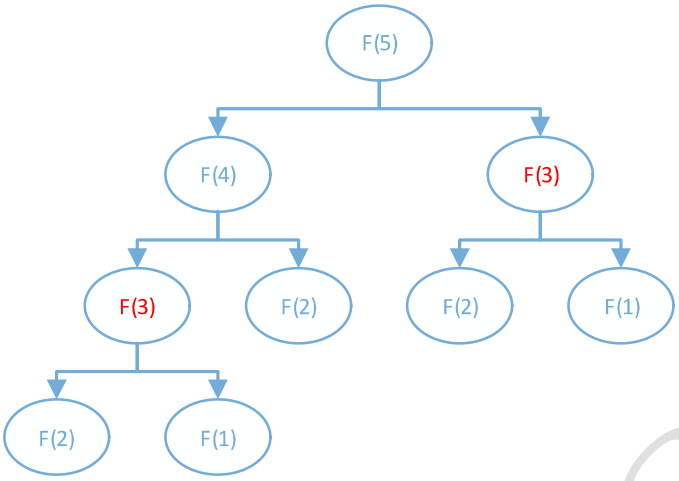
5、回到 3 楼，执行三楼还未执行的第 8 行，输出 ch。注意，这时 3 楼的 ch 的值是什么？是‘C’，所以首先输出了‘C’。3 楼的语句全部执行完成，接着 return 会 2 楼。

6、回到 2 楼，执行 2 楼还未执行的第 8 行，输出 ch。注意，2 楼的 ch 的值是什么？是‘B’。所以接着输出了‘B’。2 楼的语句全部执行，继续 return 会 1 楼。

7、回到 1 楼，执行 1 楼还未执行的第 8 行语句，输出 ch 的值‘A’，语句执行结束。继续 return，回到 main()函数。

递归的优化

在进行递归调用时，经常会对已求解过的问题进行重复求解，这造成了计算资源的浪费。比如在求斐波拉契数列时，如 f(5),求解树如下：



在上图中，我们可以看到想 f(3)被重复求过一次。如果我们要求 f(6)，那么 f(4)至少要多运算一次。求的数值越大，被重复求解的数目便越多。为了避免这种浪费，我们需要对递归函数进行优化。

1、使用记忆化进行优化

所谓记忆化，是指将求解过程中的解存储起来，在求解新问题时，判断该问题是否求解过，如果求解过，则直接返回解，不需重复求解。我们常用数组来存储求解过的解。以下代码是对斐波拉契数列递归求法的优化：

```
#include<iostream>
using namespace std;
int a[100],n;
int f(int n){
    if(n==1||n==2)return 1; //边界判断
    if(a[n])return a[n];//如果有解，直接返回结果
    else return a[n]=f(n-1)+f(n-2);//递归计算，然后存储结果，再返回结果
}
int main(){
    cin>>n;
    cout<<f(n);
}
```

2、将递归转为递推

在递归调用的过程中，由于要保留每次调用时的参数和局部变量，而且要经过逐层调用和逐层返回两大过程，程序对于时间和存储空间的耗费可以说是巨大的。因此，在一个问题对于时空要求较高的情况下，有时不得不将递归方法转化成非递归方法，这一过程，常常称之为消除递归。消除递归的过程，其本质上就是要求模拟复杂的递归工作栈的变化过程，虽然递归与非递归在形式上大多能够相互转化，但是，非递归的编程难度要远远高于递归的方法。以下是斐波拉契数列的递推写法：

```
#include<iostream>
using namespace std;
int f[100];
int main(){
    int n;
    cin>>n;
    f[0]=f[1]=1;
    for(int i=2;i<=n;i++){
        f[i]=f[i-1]+f[i-2];
    }
    cout<<f[n];
}
```

ackerman 函数

【题目描述】

计算 ackerman 函数值：

$$Ack(m,n)=\begin{cases} n+1 & m=0 \\ Ack(m-1,1) & n=0 \\ Ack((m-1),Ack(m,n-1)) & m,n>0 \end{cases}$$

【输入】

第一行为两个数，即 M 和 N，其中 0<=M<=3，0<=N<=11。

【输出】

输出 ack(m,n)的值。

【样例输入】

0 1

【样例输出】

2

【分析】

根据题目描述定义递归函数，注意边界情况的处理。

【代码】

```
#include<iostream>
using namespace std;
int n,m;
int ack(int m,int n){
    if(m==0)return n+1;
    if(n==0)return ack(m-1,1);
    return ack(m-1,ack(m,n-1));
}
int main(){
    cin>>m>>n;
    cout<<ack(m,n);
}
```

圆环套圆环

【题目描述】

一个有趣的圆环套圆环函数被定义如下：

G(n)=n-G(G(n-1)) (n 是正整数)

G(0)=0

请你计算出圆环函数的值。

【输入】

一个非负整数 n，n<=200。

【输出】

一个正整数，即 G(n)。

【样例输入】

3

【样例输出】

2

【分析】

根据题目定义递归函数，如代码 1。但是会超时。

为了避免超时，需要记忆化处理，如代码 2。

也可以转化成递推的方式处理，如代码 3。

【代码 1】

```
#include<iostream>
using namespace std;
int n;
int g(int n){
    if(n==0)return 0;
    return n-g(g(n-1));
}
int main(){
    cin>>n;
    cout<<g(n);
}
```

【代码 2】

```
#include<iostream>
using namespace std;
int n,a[201];
int g(int n){
    if(n==0)return 0;
    if(a[n])return a[n];
    else return a[n]=n-g(g(n-1));
}
int main(){
    cin>>n;
    cout<<g(n);
}
```

【代码 3】

```
#include <iostream>
using namespace std;
int n,g[201];
int main(void){
    cin>>n;
    for(int i=1;i<=n;++i)
        g[i]=i-g[g[i-1]];
    cout<<g[n];
}
```

递归函数

【题目描述】

对于一个递归函数 w(a,b,c)

如果 a<=0 or b<=0 or c<=0 就返回值 1.

如果 a>20 or b>20 or c>20 就返回 w(20,20,20)

如果 a<b 并且 b<c 就返回 w(a,b,c-1)+w(a,b-1,c-1)-w(a,b-1,c)

其它别的情况就返回 w(a-1,b,c)+w(a-1,b-1,c)+w(a-1,b,c-1)-w(a-1,b-1,c-1)

这是个简单的递归函数，但实现起来可能会有些问题。当 a,b,c 均为 15 时，调用的次数将非常的多。你要想个办法才行。

【输入】

会有若干行，并以-1，-1，-1 结束.

保证输入的数在-9223372036854775808~9223372036854775807 之间

并且是整数

【输出】

输出若干行

格式:

w(a,_b,_c)=_你的输出(_代表空格)

【样例输入】

1 1 1

2 2 2

10 4 6

50 50 50

-1 7 18

-1 -1 -1

【样例输出】

w(1, 1, 1) = 2

w(2, 2, 2) = 4

w(10, 4, 6) = 523

w(50, 50, 50) = 1048576

w(-1, 7, 18) = 1

【分析】

根据题目定义递归函数，必然超时。根据参数个数，开一个三维数组，进行记忆化处理。

【代码】

```
#include<iostream>
using namespace std;
long long a,b,c,f[21][21][21];
long long w(long long a,long long b,long long c){
    if(a<=0||b<=0||c<=0)return 1;
    if(a>20||b>20||c>20)return w(20,20,20);
    if(f[a][b][c])return f[a][b][c];
    if(a<b&&b<c)return f[a][b][c]=w(a,b,c-1)+w(a,b-1,c-1)-w(a,b-1,c);
    return f[a][b][c]=w(a-1,b,c)+w(a-1,b-1,c)+w(a-1,b,c-1)-w(a-1,b-1,c-1);
}
int main(){
    while(cin>>a>>b>>c){
        if(a==-1&&b==-1&&c==-1)break;
        cout<<"w("<<a<<", "<<b<<", "<<c<<") = "<<w(a,b,c)<<endl;
    }
}
```

分岔路口

【题目描述】

约翰的 N(1≤N≤1,000,000,000)只奶牛要出发去探索牧场四周的土地。她们将沿着一条路走，一直走到三岔路口（可以认为所有的路口都是这样的）。这时候，这一群奶牛可能会分成两群，分别沿着接下来的两条路继续走。如果她们再次走到三岔路口，那么仍有可能继续分裂成两群继续走。

奶牛的分裂方式十分古怪：如果这一群奶牛可以精确地分成两部分，这两部分的牛数恰好相差 K(1≤K≤1000)，那么在三岔路口牛群就会分裂。否则，牛群不会分裂，她们都将在这里待下去，平静地吃草。请计算，最终将会有多少群奶牛在平静地吃草。

【输入】

两个整数 N 和 K。

【输出】

最后的牛群数。

【样例输入】

6 2

【样例输出】

3

【提示】

6 只奶牛先分成 2 只和 4 只。4 只奶牛又分成 1 只和 3 只。最后有三群奶牛。

【分析】

递归模拟牛的分裂。能够分队的条件是（n-k）%2==0，即两队的差为 k 且都为整数，同时要特别注意，n-k 必须大于 0，等于 0 或小于没意义。如果可以分队，那么一队为（n+k）/2，另一队为（n-k）/2。

【代码】

```
#include<iostream>
using namespace std;
int n,k,ans;
void go(int n){
    if((n-k)%2==0&&n-k>0){  //可以分队
        go((n+k)/2);  //模拟两队的牛数递归下去
        go((n-k)/2);
    }else{
        ans++;  //不能分裂，牛群数加 1
    }
}
int main(){
    cin>>n>>k;
    go(n);
    cout<<ans;
}
```

国王的魔镜

【题目描述】

国王有一个魔镜，可以把任何接触镜面的东西变成原来的两倍——只是，因为是镜子嘛，增加的那部分是反的。比如一条项链，我们用 AB 来表示，不同的字母表示不同颜色的珍珠。如果把 B 端接触镜面的话，魔镜会把这条项链变为 ABBA。如果再用一端接触的话，则会变成 ABBAABBA（假定国王只用项链的某一端接触魔镜）。给定最终的项链，请编写程序输出国王没使用魔镜之前，最初的项链可能的最小长度。

【输入】

只有一个字符串，由大写英文字母组成（字母数<=100000），表示最终的项链。

【输出】

只有一个整数，表示国王没使用魔镜前，最初的项链可能的最小长度。

【样例输入】

ABBAABBA

【样例输出】

2

【分析】

对于字符串，如果长度为奇数，则无法二分，输出长度即可。如果长度为偶数，判断二分后左右两个字符串是否对称，如果对称，则递归二分下去，如果左右不对称，即不是翻转得来，那么也可以直接输出长度。

【代码】

```
#include<iostream>
#include<string>
using namespace std;
string s;
bool check(int k){  //判断是否回文
    for(int i=0;i<k;i++){
        if(s[i]!=s[2*k-1-i])return false;
    }
    return true;
}
void solve(int len){
    if(len%2!=0){  //如果长度为奇数，则无法二分下去
        cout<<len;
        return ;
    }
    if(check(len/2)){  //如果左右对称
        solve(len/2);  //则二分求解
    }else{
        cout<<len;  //不对称，则输出答案
        return;
    }
}
int main(){
    cin>>s;
    solve(s.size());
}
```

分形

【题目描述】

分形（fractal）是物体在数量上，内容上“自相似”的一种数学抽象。

一个盒分形（box fractal）定义如下：

? 1 度的盒分形为

X

? 2 度的盒分形为

x x

x

x x

? 如果 B(n - 1)表示 n - 1 度的盒分形，则 n 度的盒分形递归定义如下：

B(n-1) B(n-1)

B(n-1)

B(n-1) B(n-1)

请画出 n 度的盒分形的图形。

【输入】

输入由若干测试用例组成，每行给出一个不大于 7 的正整数。输入的最后一行以一个负整数-1 表示输入结束。

【输出】

对于每个测试用例，输出用'X'标记的盒分形。注意'X'是大写字母。在每个测试用例后输出包含一个单破折号的一行。

【样例输入】

1

2

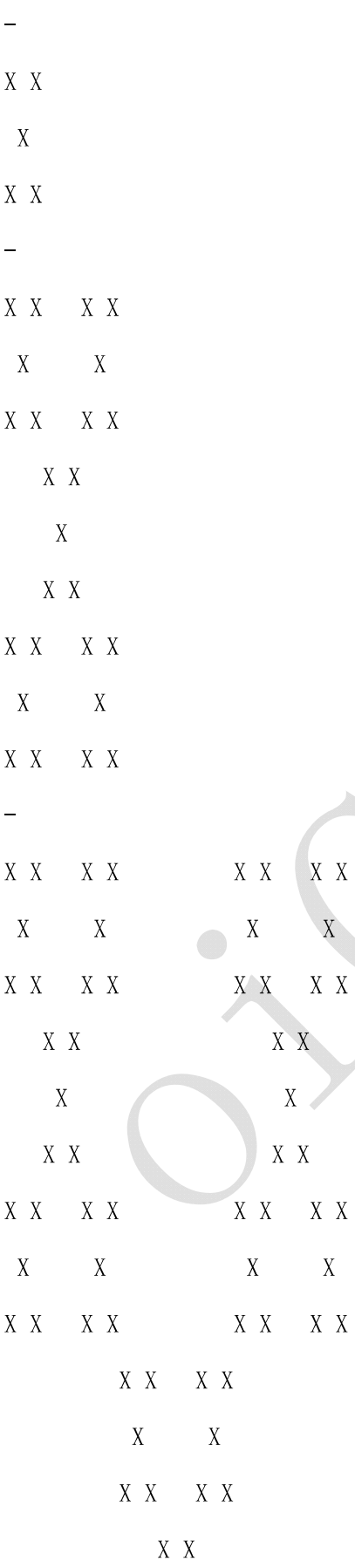
3

4

-1

【样例输出】

X



```
              X

             X X

          X X   X X

           X     X

          X X   X X

X X   X X         X X   X X

 X     X           X     X

X X   X X         X X   X X

   X X               X X

    X                 X

   X X               X X

X X   X X         X X   X X

 X     X           X     X

X X   X X         X X   X X
```

_

【分析】

利用递归思想进行图形打印。可以定义一个递归函数 paint(n,x,y)表示打印一个 n 度的分形，左上角坐标为 x，y，根据题意，一个 n 度的分形，它的宽度为 3^(n-1)。打印一个 n 度的分形，可以通过递归打印 5 个 n-1 度的分形实现，对于这 5 个 n-1 度的分形，需要仔细计算出每个分形的左上角坐标。详见代码。

【代码】

```
#include<iostream>
#include<cmath>
using namespace std;
char ch[801][801];
int n;
void paint(int n,int x,int y){
    if(n==1){  //边界条件
        ch[x][y]='X';
    }else{
        int len=pow(3,n-2);  //n-1 度的分形长度
        paint(n-1,x,y);  //左上角
        paint(n-1,x,y+2*len);  //右上角
        paint(n-1,x+len,y+len);  //中间
```

```
14.         paint(n-1,x+2*len,y);  //左下角
15.         paint(n-1,x+2*len,y+2*len);  //右下角
16.     }
17. }
18. void print(int n,int x,int y){  //输出图形
19.     int len=pow(3,n-1);
20.     for(int i=1;i<=len;i++){
21.         for(int j=1;j<=len;j++){
22.             if(ch[i][j]=='X')cout<<'X';
23.             else cout<<' ';
24.         }
25.         cout<<endl;
26.     }
27. }
28. int main(){
29.     while(cin>>n){
30.         if(n==-1)break;
31.         paint(n,1,1);
32.         print(n,1,1);
33.         cout<<"-"<<endl;
34.     }
35. }
```