算法入门

何为算法

- 解决同样的问题，我们可以有不同的方法，它们有的在效率上有差别（时间开销），有的在内存占用上有差别（空间开销）
- 比如我们熟知的问题：求数列a_1、a_2、a_3、……、a_n的和，我们当然可以循环跑一遍逐项相加，也可以借助求和公式来做
- 这都是算法：解决问题的有限步骤
- 明显的，算法有优劣之分。尤其对OIer来说，用足够优的算法解决问题，是选手在比赛中胜出的关键

时间复杂度/空间复杂度

- 在C++语言学习的过程中，我们主要关心是否能得到正确的解
- 而在接下来算法的学习中，我们会越来越多的讨论时间复杂度和空间复杂度，尤其是前者
- 一个无法在规定时间内出解的程序，与得到错误解的程序，待遇是一样的：0分

目录

- 前缀和
- 差分序列
- 递推
- 贪心
- 相遇问题
- 分治
- 逆序对
- 快速幂
- 二分
- 尺取法

算法入门

- 我们就从一个（很小 && 易于理解）的问题开始踏上算法之路

前缀和与差分序列

求数组元素前k项和

- 给定长度为n的无序数组，求其前k项的和（保证和不超int范围）

k≤n≤200,000

Sample input	Sample output
5 //n 2 1 4 8 3 3 //k	7

秒了它

- 时间复杂度O(k)

```
#include<bits/stdc++.h>
using namespace std;
int a[200010], n, k;
int main()
{
    scanf("%d%d", &n);
    for (int i = 0; i < n; i ++) scanf("%d", &a[i]);
    scanf("%d", &k);
    for (int i = 0; i < k; i ++) ans += a[i];
    printf("%d\n", ans);
    return 0;
}
```

问题升级

- 给定长度为n的无序数组，求其前k项的和（保证和不超int范围），共有m次询问

k≤n≤200,000， m≤10,000

Sample input	Sample output
5 3 //n，m 2 1 4 8 3 3 //m个k 2 4	7 3 15

继续秒

这个算法可
以说非常无脑
暴力了

- 时间复杂度O(k*m)

```
#include<bits/stdc++.h>
using namespace std;
int a[200010], n, m, k;
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i ++) scanf("%d", &a[i]);
    while (m --)
    {
        int ans = 0;
        scanf("%d", &k);
        for (int i = 0; i < k; i ++) ans += a[i];
        printf("%d\n", ans);
    }
    return 0;
}
```

存在的问题

- $k*m=2\times10^5\times10^4=2\times10^9$
- 复杂度高的主要原因是：上一次的询问结果，对于下一次询问没有任何帮助，导致每一次都要从头开始扫
- 比如一组k

300

302

100

解决的办法

- 比如一组k

300

302

100

- 如果我们可以把之前求过的和保存下来？
- 那么k=100这样的询问怎么办？又从头开始？
- 不如先把所有的和**全部**保存下来

解决的办法

- 我们另建立一个辅助数组：sum[ ]

a[0]	a[1]	a[2]	a[3]	a[4]	
2	1	4	8	3	……

- 代码实现：

```
sum[i] = sum[i-1] + a[i];
```

sum[0]	sum[1]	sum[2]	sum[3]	sum[4]	
2	3	7	15	18	……

- 这里要注意 i 的范围了

参考代码

这个算法，
有一个名字叫
前缀和

- 时间复杂度？

O(n+m)

$2\times10^5+10^4\approx2\times10^5$

```
#include<bits/stdc++.h>
using namespace std;
int a[200010], sum[200010], n, m, k;
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++)  //不能从 0开始
    {
        scanf("%d", &a[i]);
        sum[i] = sum[i-1] + a[i];
    }
    while (scanf("%d", &k) == 1)
        printf("%d\n", sum[k]);
    return 0;
}
```

效率对比

- 测试数据规模n=200,000

试题 选手

名称	排名	preand	总分	总用时(s)	测试时间
前缀和	1	100	100	0.468	2019/1/9 23:02:56
无脑暴力	1	100	100	7.89	2019/1/9 23:03:58

前缀和/预处理

- 前缀和通过复杂度O(n)的预处理，实现复杂度O(1)的查询
- 独立的前缀和问题是比较少的，因为考得太浅。它更多的是成为解决一个问题的完整算法中的一部分：用作统计答案前的**预处理**

拓展一下

- 如果我们每次问的不是前k项的和，而是指定区间[l, r]的和呢？
- 同样是O(1)的查询：

```
printf("%d", sum[r] - sum[l-1]);
```

a[0]	a[1]	a[2]	a[3]	a[4]	
2	1	4	8	3	……

sum[0]	sum[1]	sum[2]	sum[3]	sum[4]	
2	3	7	15	18	……

l r

组合数问题

组合数 C_n^m 表示的是从 n 个物品中选出 m 个物品的方案数。举个例子，从 $(1,2,3)$ 三个物品中选择两个物品可以有 $(1,2),(1,3),(2,3)$ 这三种选择方法。根据组合数的定义，我们可以给出计算组合数 C_n^m 的一般公式：

$$C_n^m = \frac{n!}{m!(n-m)!}$$

其中 $n! = 1 \times 2 \times \cdots \times n$。

小葱想知道如果给定 n, m 和 k，对于所有的 $0 \le i \le n, 0 \le j \le \min(i, m)$ 有多少对 (i, j) 满足 C_i^j 是 k 的倍数。

Sample Input	Sample Output	Sample Input2	Sample Output2
1 2 //测试组数t、k 3 3 //n、m	1	2 5 4 5 6 7	0 7

分析

- 样例1相当于问：c_0^0、c_1^0、c_1^1、 c_2^0、 c_2^1、 c_2^2、c_3^0、c_3^1、 c_3^2、 c_3^3中，其计算结果有多少个是2的倍数。注意：

1. i、j都是可以取到0的，而$c_x^0 \neq 0$，所以不能忽略
2. j≤i

- 样例2，大家自己推一下

分析

- 样例1相当于问：c_0^0、c_1^0、c_1^1、 c_2^0、 c_2^1、 c_2^2、c_3^0、c_3^1、 c_3^2、 c_3^3中，其计算结果有多少个是2的倍数。

- 于是我们很无脑地去一项项算阶乘，并代入计算？ $C_n^m = \frac{n!}{m!(n-m)!}$
- 如果这样（程序）都不爆，那太没天理了

分析

- 样例1相当于问：c_0^0、c_1^0、c_1^1、 c_2^0、 c_2^1、 c_2^2、c_3^0、c_3^1、 c_3^2、 c_3^3中，其计算结果有多少个是2的倍数。

$$C_n^m = \frac{n!}{m!(n-m)!}$$

- 我们肯定不能老老实实去算阶乘
- 可是c_n^m又必须要计算
- 有个著名的组合数公式如下：

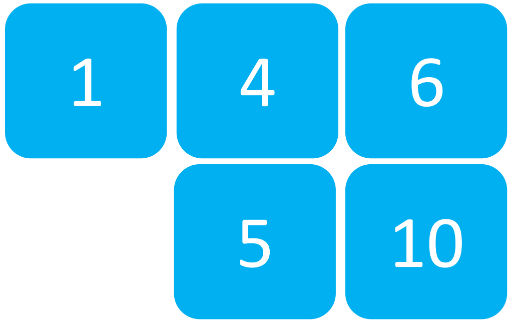
$$\boldsymbol{c_n^m = c_{n-1}^{m-1} + c_{n-1}^m}$$

分析

- 样例1相当于问：c_0^0、c_1^0、c_1^1、 c_2^0、 c_2^1、 c_2^2、c_3^0、c_3^1、 c_3^2、 c_3^3中，其计算结果有多少个是2的倍数。

- 有个著名的组合数公式如下：

$$c_n^m = c_{n-1}^{m-1} + c_{n-1}^m$$



- 联想到著名的杨辉三角，就是这个递推式

分析

- 于是转化为求n行的杨辉三角中，有多少项是k的倍数
- 杨辉三角我们之前在二维数组中已经写过了，大家都会
- 但是这题数据范围$n \le 2000$，$t \le 10000$，还是会超时
- 这个问题相当于：t组询问，每次问一个二维数组从左上角开始的符合条件的项有多少个，这是统计答案个数

二维前缀和

- 前缀和的思想是可以推广到二维的

a[ ][ ]

2	1	8	9	4
0	4	-2	7	10
3	11	2	3	4
4	6	5	-3	1
15	2	4	5	-8

sum[ ][ ]

2	3	11	20	24
2	7	13	29	43
5	21	29	48	66
9	31	44	60	79
24	48	65	86	97

- 递推式： **sum[i][j] = sum[i-1][j] + sum[i][j-1] – sum[i-1][j-1] + a[i][j]**

回来

- 于是我们对杨辉三角也干同样的事：建一个对应的二维数组sum[i][j]，用于统计和
- 但这个和对我们没用

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1

1				
2	3			
3	6	7		
4	10	14	15	
5	15	25	30	31

回来

- 于是我们对杨辉三角也干同样的事：建一个对应的二维数组sum[i][j]，用于统计和
- 我们只统计能被k整除的项的个数和

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1

1				
1	1			
1	0	1		
1	3	3	1	
1	0	0	0	1

0				
0	0			
0	1	1		
0	1	1	1	
0	2	3	4	4

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
long long f[2010][2010], sum[2010][2010];  //要开long long
int t, k, n, m;
int main()
{
    scanf("%d%d", &t, &k);
    for (int i = 0; i <= 2000; i ++) f[i][i] = f[i][0] = 1;  //初始化
    for (int i = 1; i <= 2000; i ++)
        for (int j = 1; j <= 2000; j ++)
        {
            f[i][j] = (f[i-1][j] + f[i-1][j-1]) % k;
            sum[i][j] = sum[i][j-1] + sum[i-1][j] - sum[i-1][j-1];  //二维前缀和
            if (f[i][j] == 0 && j <= i) sum[i][j] ++;
        }
    while (t --)
    {
        scanf("%d%d", &n, &m);
        printf("%d\n", sum[n][m]);
    }
    return 0;
 }
```

区间修改

- 给定长度为n的无序数组，对它进行m次修改操作
- 修改的格式为：l r k　意思即为对数组[l,r]范围内的元素全部加上值k，最后要求输出经过这m次修改后的数组

l≤r≤n≤200,000，m≤10,000，k∈int，不保证非负

Sample input	Sample output
10 5 //n，m 2 1 8 9 4 15 7 11 -3 6 //a_i 2 6 2 //l、r、k，下同 5 9 -3 7 8 5 0 2 1 1 3 1	3 3 12 12 6 14 6 13 -1 3

暴力秒

```
10 5
2 1 8 9 4 15 7 11 -3 6
2 6 2
5 9 -3
7 8 5
0 2 1
1 3 1
3 3 12 12 6 14 6 13 -1 3
```

- 复杂度O(n*m)

```
#include<bits/stdc++.h>
using namespace std;
int a[200010], n, m, l, r, k;
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < n; i ++) scanf("%d", &a[i]);
    while (m --)
    {
        scanf("%d%d%d", &l, &r, &k);
        for (int i = l; i <= r; i ++) a[i] += k;
    }
    for (int i = 0; i < n; i ++) printf("%d ", a[i]);
    return 0;
}
```

分析

- 复杂度高的原因，依然在于需要逐段扫描[l,r]
- 但是这题的需求是[l,r]中的每一个元素都要修改，看起来只能逐项修改，难道还可以跳跃式修改吗？

a[0]	a[1]	a[2]	a[3]	a[4]	
2	1	4	8	3	……

分析

- 不妨再研究下前缀和
- 假如a[1]号元素被修改了：+2，那么对应的sum数组有什么变化？

a[0]	a[1]	a[2]	a[3]	a[4]	
2	1	4	8	3	……

a[0]	a[1]	a[2]	a[3]	a[4]	
2	3	4	8	3	……

sum[0]	sum[1]	sum[2]	sum[3]	sum[4]	
2	3	7	15	18	……

sum[0]	sum[1]	sum[2]	sum[3]	sum[4]	
2	5	9	17	20	……

分析

- 不妨再研究下前缀和
- 假如修改a[1]~a[3]号元素：+2，那么对应的sum数组又有什么变化？

24-18=6

a[0]	a[1]	a[2]	a[3]	a[4]	
2	1	4	8	3	……

a[0]	a[1]	a[2]	a[3]	a[4]	
2	3	6	10	3	……

sum[0]	sum[1]	sum[2]	sum[3]	sum[4]	
2	3	7	15	18	……

sum[0]	sum[1]	sum[2]	sum[3]	sum[4]	
2	5	11	21	24	……

分析

- 如果我们在a[1]号位置+2，在a[4]号位置-2
- 我们在sum[1]~sum[3]的每个元素的值就都实现了+2
- 意义在哪里？我们**通过O(1)的操作，实现了O(n)的区间修改**

a[0]	a[1]	a[2]	a[3]	a[4]	
2	1	4	8	3	……

sum[0]	sum[1]	sum[2]	sum[3]	sum[4]	
2	3	7	15	18	……

a[0]	a[1]	a[2]	a[3]	a[4]	
2	3	4	8	1	……

sum[0]	sum[1]	sum[2]	sum[3]	sum[4]	
2	5	9	17	18	……

分析

- 等等，我们要实现区间修改的是a数组！
- 那简单，我们就让a数组成为某个数组的前缀和，然后对那个数组做O(1)操作就行了
- b数组是a数组的“逆前缀和”，修改b[1]+2，b[4]-2

a[0]	a[1]	a[2]	a[3]	a[4]	
2	1	4	8	3	……

b[0]	b[1]	b[2]	b[3]	b[4]	
2	-1	3	4	-5	……

b[0]	b[1]	b[2]	b[3]	b[4]	
2	1	3	4	-7	……

分析

这个算法，
也有一个名字
叫**差分序列**

- b数组是a数组的“逆前缀和”，修改b[1]+2，b[4]-2
- 然后再对b数组求前缀和，还原出a数组，顺利实现区间修改

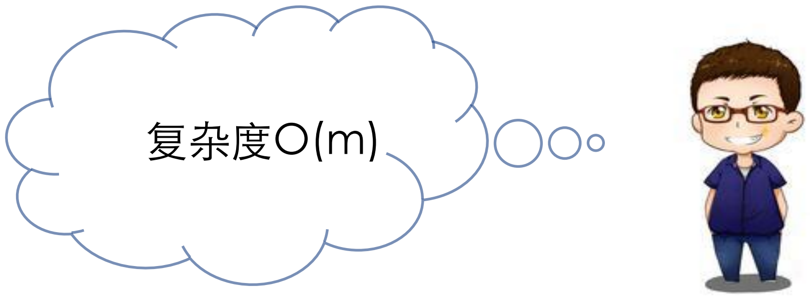
a[0]	a[1]	a[2]	a[3]	a[4]	
2	1	4	8	3	……

a[0]	a[1]	a[2]	a[3]	a[4]	
2	3	6	10	3	……

b[0]	b[1]	b[2]	b[3]	b[4]	
2	-1	3	4	-5	……

b[0]	b[1]	b[2]	b[3]	b[4]	
2	1	3	4	-7	……

参考代码




```
#include<bits/stdc++.h>
using namespace std;
int a[200010], b[200010], n, m, l, r, k;
int main()
{
    scanf("%d%d", &n, &m);
    scanf("%d", &a[0]); b[0] = a[0];  //单独处理首个元素
    for (int i = 1; i < n; i ++)
        scanf("%d", &a[i]), b[i] = a[i] - a[i-1];  //生成 b数组
    while (m --)
    {
        scanf("%d%d%d", &l, &r, &k);
        b[l] += k; b[r+1] -= k;
    }
    for (int i = 0; i < n; ++i)
        a[i] = a[i-1] + b[i], printf("%d ", a[i]);  //还原 a数组并输出
    return 0;
 }
```

效率对比

- 测试数据规模n=200,000， m=10,000

试题 选手

名称	排名	difference	总分	总用时(s)	测试时间
差分序列	1	100	100	0.546	2019/1/14 16:23:59
暴力大法	1	100	100	2.25	2019/1/14 16:23:53

递推策略

斐波拉契数列

- 求数列1、1、2、3、5、8、13、……的第n项

- 这题我们之前在介绍迭代思想的时候就做过了

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int i, n, a2 = 1, a1 = 1, a = 1;
    cin >> n;
    for (i = 3; i <= n; i ++)
    {
        a = a2 + a1;
        a2 = a1;    //迭代
        a1 = a;     //迭代
    }
    cout << a;
    return 0;
}
```

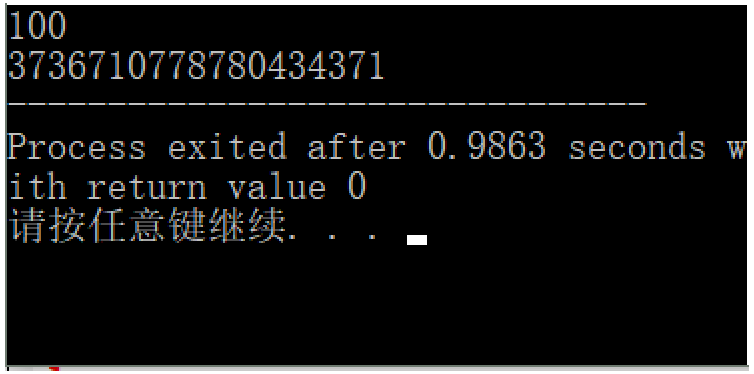
斐波拉契数列

- 我们设f(n)表示斐波拉契数列的第n项，有

$$f(n)=f(n-1)+f(n-2)$$

- 于是我们可以写出新的代码：

新代码



- 复杂度O(n)

```
#include<bits/stdc++.h>
using namespace std;
long long a[1010];
int main()
{
    int n; scanf("%d", &n);
    a[1] = a[2] = 1;
    for (int i = 3; i <= n; i ++)
        a[i] = a[i-1] + a[i-2];
    printf("%lld", a[n]);
}
```

递推

- 如果数列中的某项，与它前面的或者后面的若干项有某种关联，可以用一定的代数式表达出来
- 求解的时候，我们从初始的某项出发，通过代数关系推导计算出未知项的做法，即为递推
- 递推既有顺推，也有逆推

- 斐波拉契数列的完整递推式：

1. **f(1)=f(2)=1**
2. **f(n)=f(n-1)+f(n-2) (n>2)**

走楼梯

- 楼梯有n阶台阶，上楼可以一步走一阶，也可以一步走两阶。编程输出要走到第n阶台阶，一共有多少种不同的走法

分析

- 考虑最后一步，要么从n-1阶跨上来，要么从n-2阶跨上来，没有第三种情况
- 设f(n)表示走完n阶台阶的不同走法数

1. **f(1)=1**
2. **f(2)=2**
3. **f(n)=f(n-1)+f(n-2) (n>2)**

- 这是斐波拉契数列的经典应用

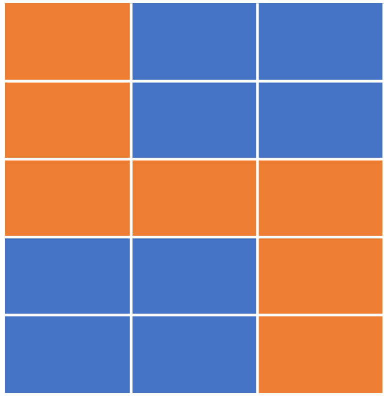
参考代码

```
100
12987777288209840005
--------------------------------
Process exited after 2.334 seconds wi
th return value 0
请按任意键继续. . .
```

```
#include<bits/stdc++.h>
using namespace std;
long long a[1010];
int main()
{
    int n; scanf("%d", &n);
    a[1] = 1; a[2] = 2;
    for (int i = 3; i <= n; i ++)
        a[i] = a[i-1] + a[i-2];
    printf("%lld", a[n]);
    return 0;
}
```

铺地板

- 有白色1×1和黑色2×2两种不同规格的木地板，要铺满n×3的地面。编程输出共有多少种铺设方案

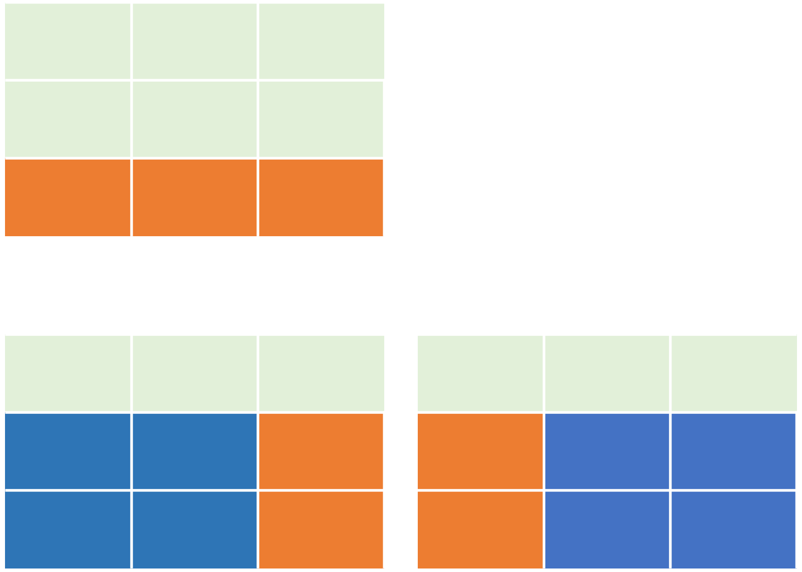


分析

- 考虑最后一步的情况，要么用三块1×1的铺满，要么用1块2×2的，两块1×1的铺满，没有第四种情况

- 设f(n)表示铺设n×3的地面的方案数

1. $f(1)=1$
2. $f(2)=3$
3. $f(n)=f(n-1)+f(n-2)\times 2 \quad (n>2)$



参考代码

```
50
28143753123
--------------------------------
Process exited after 1.4 seconds with
 return value 0
请按任意键继续. . .
```

```
#include<bits/stdc++.h>
using namespace std;
long long a[1010];
int main()
{
    int n; scanf("%d", &n);
    a[1] = 1; a[2] = 3;
    for (int i = 3; i <= n; i ++)
        a[i] = a[i-1] + a[i-2];
    printf("%lld", a[n]);
}
```

数的计算

- 输入一个自然数n（n≤1000），然后对此自然数按照如下方法进行处理

1. 不作任何处理
2. 在它的左边加上一个自然数，但该自然数不能超过原数的一半
3. 加上数后，继续按此规则进行处理，直到不能再加自然数为止

- 你的程序要能统计出符合这些条件的数的个数

Sample input	Sample output
6 //n	6
解释：6、16、26、126、36、136	

分析

- 这题我们在学习递归时布置过课外加练
- 可惜因为超时被卡成了25分


```
#include<bits/stdc++.h>
using namespace std;
int ans, n;
int cnt(int x)
{
    ans ++;
    for (int i = 1; i <= x / 2; i ++)
        cnt(i);
    return ans;
}
int main()
{
    scanf("%d", &n);
    printf("%d", cnt(n));
    return 0;
 }
```

分析

- 我们设f(n)表示初始自然数n的符合条件的数的个数
- 直接很难得到规律，我们打个表看一看

f(1)=1 //1

f(2)=f(1)+1 //2，12

因为3的一半最多能到1，所以f(3)=f(1)+1 //3，13

因为4的一半可以到2了，所以f(4)=f(1)+f(2)+1 //4，24，124，14

因为5的一半最多能到2，所以f(5)=f(1)+f(2)+1 //5，25，125，15

因为6的一半可以到3了，所以f(6)=f(1)+f(2)+f(3)+1 //6，36，136，26，126，16

分析

- 于是我们根据n的奇偶性作出讨论：

1. n是偶数，把n/2的方案数累加起来

```
if (i % 2 == 0)
    for (int j = 1; j <= i / 2; j ++)
        a[i] += a[j];
```

2. n是奇数，把它等同于前一个偶数n-1

```
if (i % 2 != 0) a[i] = a[i-1];
```

- 最后不要忘记加上n自己

参考代码

```
#include<bits/stdc++.h>
using namespace std;
int a[1010], n;
int main()
{
    scanf("%d", &n);
    a[1] = 1;
    for (int i = 2; i <= n; i ++)
        if (i % 2 == 0)
        {
            for (int j = 1; j <= i / 2; j ++)
                a[i] += a[j];
            a[i] ++;  //加上自己
        }
        else a[i] = a[i-1];
    printf("%d", a[n]);
    return 0;
}
```

测试点信息

#1 AC 0ms/2058KB	#2 AC 0ms/2117KB	#3 AC 0ms/2117KB	#4 AC 0ms/2105KB	#5 AC 0ms/2058KB	#6 AC 0ms/2109KB	#7 AC 0ms/2027KB
#8 AC 0ms/2027KB	#9 AC 0ms/2109KB	#10 AC 0ms/2078KB	#11 AC 0ms/1949KB	#12 AC 0ms/2121KB	#13 AC 0ms/2031KB	#14 AC 0ms/2113KB
#15 AC 0ms/2046KB	#16 AC 0ms/2128KB	#17 AC 0ms/2023KB	#18 AC 0ms/2003KB	#19 AC 0ms/2031KB	#20 AC 0ms/2058KB	

递推不要
忽略初始值

数的划分

- 将自然数n分成k份（不能为零），问有多少种不同的划分方式
- 注意：1、2和2、1这样的被认为是同一种划分

6≤n≤200， 2≤k≤6

Sample input	Sample output
7 3 //n，k	4
解释：1/1/5，1/2/4，1/3/3，2/2/3	

分析

- 这题咋看是没什么思路的，搜索还没学

- 我们可以考虑n不同时的划分方案之间有什么递推关系
- 设f[n][k]表示把n分成k份的划分方案数
- 我们假设分出了至少一个1

$$f[n][k]=f[n-1][k-1]$$

- 注意分出的哪一份是1，不影响答案，因为与顺序无关

分析

- 我们假设分出了至少一个1

$$f[n][k]=f[n-1][k-1]$$

- 然后有可能无法分出1，这种情况和把每个划分结果-1是一样的

$$f[n][k]=f[n-k][k]$$

- 最后，无论n是多少，只要k=1，f[n][k]=1

分析

- 得到递推式：

1. **f[n][1]=1**
2. **f[n][k]=f[n-1][k-1]+f[n-k][k] （k>1 && k <= n）**

参考代码

```
100 5
38225
--------------------------------
Process exited after 1.673 seconds wi
th return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int f[210][10], n, k;
int main()
{
    scanf("%d%d", &n, &k);
    for (int i = 1; i <= n; i ++) f[i][1] = 1;  //初始化
    for (int i = 2; i <= n; i ++)
        for (int j = 2; j <= k; j ++)
            if (j <= i) f[i][j] = f[i-1][j-1] + f[i-j][j];
    printf("%d", f[n][k]);
    return 0;
 }
```

递归

- 然后这道题也可以利用递归的思路求解

- 递归边界

k==1

- 递归方向

n-i && k-1　（i为分出来的数）

递归

- 递归边界

k==1

- 递归方向

n-i && k-1 （i为分出来的数）

- 然后有个最大的问题没有解决：当我们分出来1/2/4之后，不再分出1/4/2这样的结果

递归

- 然后有个最大的问题没有解决：当我们分出来1/2/4之后，不再分出1/4/2这样的结果

- 我们可以在函数调用时增加一个参数x，用于记录上次分出来的数

```
void cal(int n, int k, int x)  //x表示上一次分出来的数
{

}
```

递归

- 然后有个最大的问题没有解决：当我们分出来1/2/4之后，不再分出1/4/2这样的结果

- 我们可以在函数调用时增加一个参数x，用于记录上次分出来的数
- 而在下次分数尝试时，直接从x开始，这样就不可能再分出比x小的数，但1/1/5这样的不受影响

```
void cal(int n, int k, int x)  //x表示上一次分出来的数
{
    for (int i = x; i <= n / k; i ++)
    {

    }
}
```

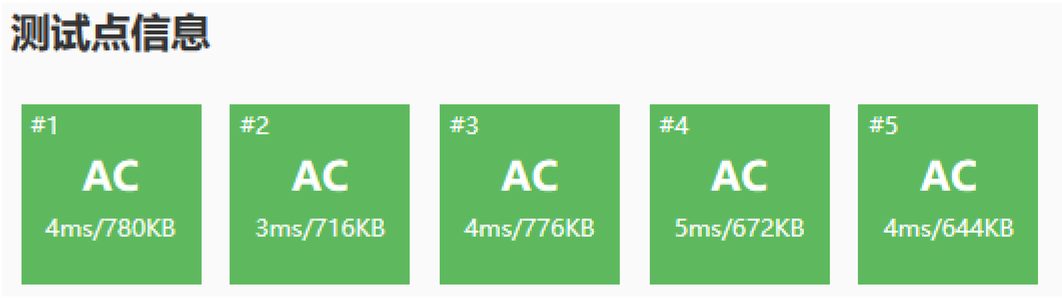
参考代码

```
100 5
38225
--------------------------------
Process exited after 1.822 seconds wi
th return value 0
请按任意键继续. . .
```

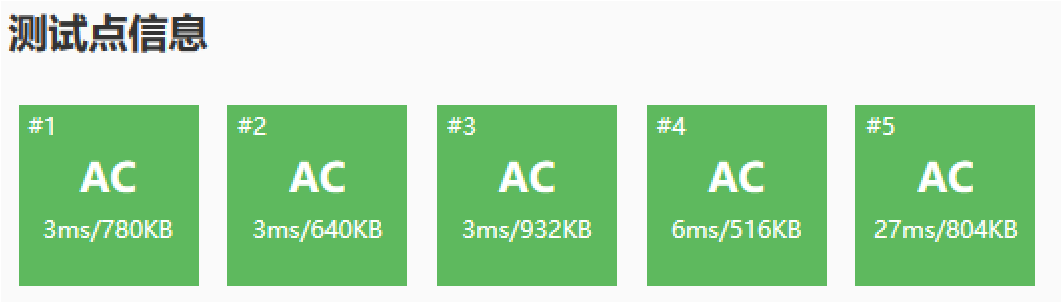
```
#include<bits/stdc++.h>
using namespace std;
int n, k, ans;
void cal(int n, int k, int x)  //x表示上一次分出来的数
{
    if (k == 1) {ans ++; return;}
    for (int i = x; i <= n / k; i ++)
        cal(n - i, k - 1, i);
}
int main()
{
    scanf("%d%d", &n, &k);
    cal(n, k, 1);
    printf("%d", ans);
    return 0;
 }
```

效率对比

- 递推解法



- 递归解法



小结

递推	递归
要有初始项	要有递归边界
要有明确的递推关系	要有明确的递归方向
单向递进	单向递进+回溯返回

- 和递归相比，递推的效率要更高
- 但是在思维强度上，递推更大

传球游戏

- n位同学站成一个圆圈，其中的一位同学手里拿着一个球。每位同学可以把球传给自己左右的两位同学中的一个（左右任意）
- 问有多少种不同的传球方法可以使得从某同学手里开始传的球，传了m次以后，又回到他自己手里。注意1-2-3-1和1-3-2-1视为不同的传球方法

Sample input	Sample output
3 3 //n，m	2

分析

- 既然是递推，就要想到某个项和其相邻项的关系
- n号同学手里的球，只可能来自于他左边的n-1号同学，或者右边的n+1号同学
- 我们设f[n][m]表示球经过了m次传递以后，传到n号同学手中的方案数

f[n][m]=f[n-1][m-1]+f[n+1][m-1]

- 这个式子表示什么意思呢？就是n-1号（左边）同学拿到球的方案数，加上n+1号（右边）同学拿到球的方案数之和

分析

- 于是初始值自然就是f[1][0]=1，最终答案就是f[1][m]

- 但是还有环的问题没有处理，因为1号和n号是处在环的结合处的，所以1号和n号同学要单独处理：

f[1][m]=f[2][m-1]+f[n][m-1]   //1号的左右两边是2号和n号

f[n][m]=f[1][m-1]+f[n-1][m-1] //n号的左右两边是1号和n-1号

参考代码

```
20 10
252
--------------------------------
Process exited after 2.018 seconds with return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int f[50][50], n, m;
int main()
{
    scanf("%d%d", &n, &m);
    f[1][0] = 1;
    for (int i = 1; i <= m; i ++)
    {
        f[1][i] = f[2][i-1] + f[n][i-1];  //1号同学
        for (int j = 2; j <= n - 1; j ++)
            f[j][i] = f[j-1][i-1] + f[j+1][i-1]; //2~n-1号同学
        f[n][i] = f[1][i-1] + f[n-1][i-1];  //n号同学
    }
    printf("%d", f[1][m]);
    return 0;
}
```

约瑟夫问题

- 大家还记得我们学循环时候的约瑟夫问题吧
- 之前的解法模拟出圈的整个过程，复杂度是O(nm)的
- 要是n、m比较大，出解就太慢了

Sample input	Sample output
7 3 //n，m	4 //最后那个人

分析

- 要应用递推的思路，就需要找出前后项之间的联系
- 我们设f[n]表示当环的人数为n时的最后胜利者的位置
- 我们将这n个人编号为**0~n-1**
- 之所以不是1~n，是因为在循环后期，m是有可能大于n的，出列的人编号m%n，而m%n有可能等于0，编号从0开始便于推导

分析

- 那么第一个出列的人编号是多少呢？

  m%n-1

- 第一个人出列之后，他后面的人，包括那个最后的胜利者，位置都相当于向前移动了m位（减去m），这样构成一个编号从m%n开始的新环

分析

- 反过来，如果我们知道f[n-1]时胜利者的位置，那么f[n]时的胜利者就往回退m位（加上m）

$$f[n]=f[n-1]+m$$

- 考虑到这是一个环，因此还需要取模

$$f[n]=(f[n-1]+m)\%n$$

分析

- 最后得到递推式

1. **f[1]=0** //第一个人编号0
2. **f[n]=(f[n-1]+m)%n (n>1)**

注意这里%n的n是随着环人数变化不断变化的，非常量

```
1000 12
214
--------------------------------
Process exited after 2.786 seconds wi
th return value 0
请按任意键继续. . .
```

参考代码

- 逐级递推，不用保存中间结果，不需要数组

```
#include<bits/stdc++.h>
using namespace std;
int n, m, ans;
int main()
{
	scanf("%d%d", &n, &m);
	for (int i = 2; i <= n; i ++)
		ans = (ans + m) % i;
	printf("%d", ans + 1);  //+1是回到编号1~n
	return 0;
 }
```

效率对比

- 测试规模n=200,000， m=10,000

试题 选手

名称	排名	josephu	总分	总用时(s)	测试时间
循环模拟	1	100	100	50.234	2019/1/15 18:03:43
递推法	1	100	100	0.015	2019/1/15 18:03:45

课外加练

- luogu 2822　组合数问题
- luogu 2038　无线网络发射器选址
- luogu 1028　数的计算
- luogu 1025　数的划分
- luogu 1057　传球游戏
- luogu 1255　数楼梯（递推+高精度）
- luogu 1096　Hanoi双塔（递推+高精度）