|  |
| --- |
| HANGZHOU NORMAL UNIVERSITY |
| ALGORITHM\_TEMPLATE |
| FOR ACM\_ICPC |

|  |
| --- |
| Origin: Zhejiang University ACM Template |
| Updated by stultus |
| 2017-04-06 |

目录

[一、 基本语法和用法 1](#_Toc479252821)

[1.1 c++输入输出的加速问题 1](#_Toc479252822)

[1.2 模板代码头 1](#_Toc479252823)

[1.3 格式化输入输出: 1](#_Toc479252824)

[1.3.1 忽略变量（%\*?） 1](#_Toc479252825)

[1.3.2 扫描集（%[]） 1](#_Toc479252826)

[1.3.3 输出前面带有0 1](#_Toc479252827)

[1.3.4 %n 2](#_Toc479252828)

[1.3.5 getline() 用法 2](#_Toc479252829)

[1.3.6 C++输出格式 2](#_Toc479252830)

[1.4 algorithm 的常用函数 2](#_Toc479252831)

[1.4.1 fill 2](#_Toc479252832)

[1.4.2 next\_permutation 2](#_Toc479252833)

[1.4.3 reverse 2](#_Toc479252834)

[1.4.4 sort 2](#_Toc479252835)

[1.4.5 max\_element / min\_element 2](#_Toc479252836)

[1.4.6 map 2](#_Toc479252837)

[1.4.7 lower\_bound() ,upper\_bound() 2](#_Toc479252838)

[1.4.8 binary\_search (a.begin(),a,end(),num) 3](#_Toc479252839)

[1.4.9 includes() 判断某区间内是否都涵盖于另一区间中 3](#_Toc479252840)

[1.5 其他 3](#_Toc479252841)

[1.5.1 避免对浮点数进行 == 或 ！= 比较 3](#_Toc479252842)

[1.5.2 取非负数 3](#_Toc479252843)

[1.5.3 各类型长度 3](#_Toc479252844)

[1.5.4 随机数 3](#_Toc479252845)

[二、 数学概念 3](#_Toc479252846)

[2.1 PI精确赋值 3](#_Toc479252847)

[2.2 cmath常用函数 3](#_Toc479252848)

[2.3 最大公约数 3](#_Toc479252849)

[2.4 快速幂 4](#_Toc479252850)

[2.5 isPrime-判断素数 4](#_Toc479252851)

[2.6 筛选法求素数（） 4](#_Toc479252852)

[2.7 组合函数C(n, r) 4](#_Toc479252853)

[2.7.1 基本用法 4](#_Toc479252854)

[2.7.2 性质 5](#_Toc479252855)

[2.7.3 应用 5](#_Toc479252856)

[2.7.4 lucas定理 5](#_Toc479252857)

[2.8 斐波那契数列性质 6](#_Toc479252858)

[2.9 约瑟夫环 6](#_Toc479252859)

[2.10 阶乘最后非0位 6](#_Toc479252860)

[2.11 数学公式 6](#_Toc479252861)

[2.11.1 勾股数 6](#_Toc479252862)

[2.11.2 n条直线最多划分成n\*(n+1)/2 + 1个平面 7](#_Toc479252863)

[2.11.3 求和公式 7](#_Toc479252864)

[2.11.4 Sn=1+1/2+1/3+…+1/n 7](#_Toc479252865)

[2.11.5 判断n!是否能够被m整除 7](#_Toc479252866)

[2.11.6 因子和的计算方法 7](#_Toc479252867)

[2.11.7 哥德巴赫猜想 7](#_Toc479252868)

[2.11.8 一绳长L，一蚂蚁从绳的一段爬向另一端，速度为v m/s，同时绳子以每秒u米的速度均匀伸长，问： 蚂蚁能否到达绳的另一端？如能，需要多长时间？ 7](#_Toc479252869)

[2.12 矩阵乘法 7](#_Toc479252870)

[2.13 模的性质 8](#_Toc479252871)

[2.13.1 基本性质 8](#_Toc479252872)

[2.13.2 运算规则 8](#_Toc479252873)

[2.13.3 重要定理 8](#_Toc479252874)

[2.14 数字取整性质 8](#_Toc479252875)

[2.15 欧拉函数 8](#_Toc479252876)

[2.16 三分法求函数极值 9](#_Toc479252877)

[三、 位运算 9](#_Toc479252878)

[四、 日期 9](#_Toc479252879)

[4.1 判断是否为闰年 9](#_Toc479252880)

[4.2 蔡勒公式 10](#_Toc479252881)

[4.3 计算日期相差天数 10](#_Toc479252882)

[4.4 计算日期相差天数（封装成函数） 11](#_Toc479252883)

[五、 字符串相关 11](#_Toc479252884)

[5.1 string常用库函数 11](#_Toc479252885)

[5.1.1 string类的构造函数 11](#_Toc479252886)

[5.1.2 string类的字符操作 11](#_Toc479252887)

[5.1.3 string的特性描述 11](#_Toc479252888)

[5.1.4 string类的输入输出操作 12](#_Toc479252889)

[5.1.5 string的赋值 12](#_Toc479252890)

[5.1.6 string的连接 12](#_Toc479252891)

[5.1.7 string的比较 12](#_Toc479252892)

[5.1.8 string的子串 12](#_Toc479252893)

[5.1.9 string的交换 12](#_Toc479252894)

[5.1.10 string类的查找函数 12](#_Toc479252895)

[5.1.11 string类的替换函数 12](#_Toc479252896)

[5.1.12 string类的插入函数 12](#_Toc479252897)

[5.1.13 string类的删除函数 13](#_Toc479252898)

[5.1.14 string类的迭代器处理 13](#_Toc479252899)

[六、 高精度算法与进制转换 13](#_Toc479252900)

[6.1 十进制转m进制 13](#_Toc479252901)

[6.2 N进制转换十进制 13](#_Toc479252902)

[6.3 长数字相乘 13](#_Toc479252903)

[6.4 高精度加减乘除取余运算 14](#_Toc479252904)

[6.5 数组方式实现的高精度与整型加法 17](#_Toc479252905)

[七、 计算几何 18](#_Toc479252906)

[7.1 几何公式 18](#_Toc479252907)

[7.2 判断一个点是否在三角形内 19](#_Toc479252908)

[7.3 判断点是否在线段上 19](#_Toc479252909)

[7.4 Graham求凸包（已修改） 20](#_Toc479252910)

[7.5 }点到线段的最短距离 20](#_Toc479252911)

[7.6 空间两线段的最短距离 20](#_Toc479252912)

[7.7 空间点到线段的最短距离 21](#_Toc479252913)

[八、 数据结构 22](#_Toc479252914)

[8.1 二分查找 22](#_Toc479252915)

[8.2 并查集 22](#_Toc479252916)

[8.3 树状数组 22](#_Toc479252917)

[8.4 线段树(部分修改) 23](#_Toc479252918)

[8.4.1 线段树的结构体和命令行（已修改） 23](#_Toc479252919)

[8.4.2 线段树的建树（已修改） 23](#_Toc479252920)

[8.4.3 线段树区间查询（已修改） 23](#_Toc479252921)

[8.4.4 线段树的区间更新（已修改） 24](#_Toc479252922)

[8.4.5 线段树的区间合并 24](#_Toc479252923)

[8.4.6 二维线段树 26](#_Toc479252924)

[8.5 拓扑排序 27](#_Toc479252925)

[8.6 单调队列 28](#_Toc479252926)

[8.7 AVL Tree (链表形式) 28](#_Toc479252927)

[九、 图论 30](#_Toc479252928)

[9.1 Dijkstra 30](#_Toc479252929)

[9.2 Floyd 31](#_Toc479252930)

[9.3 传递闭包判断图的连通性 31](#_Toc479252931)

[9.4 树的最小支配集 31](#_Toc479252932)

[9.5 Hash 32](#_Toc479252933)

[9.6 匈牙利二分图最大匹配 32](#_Toc479252934)

[9.7 并查集 32](#_Toc479252935)

[9.8 最小生成树 33](#_Toc479252936)

[9.8.1 prim算法 33](#_Toc479252937)

[9.8.2 kruskal算法 34](#_Toc479252938)

[9.9 tarjan算法找割点/割边 35](#_Toc479252939)

[十、 动态规划 35](#_Toc479252940)

[10.1 0-1背包 35](#_Toc479252941)

[10.2 一个序列最少能被划分成多少个递增子序列 36](#_Toc479252942)

[10.3 最长递增子序列 36](#_Toc479252943)

[10.4 最长公共子序列 36](#_Toc479252944)

[10.5 最长回文子序列 36](#_Toc479252945)

[十一、 自家模板 37](#_Toc479252946)

[11.1 KMP模板 37](#_Toc479252947)

# 基本语法和用法

## c++输入输出的加速问题

1.将endl替换为’\n’

2.解除绑定

cin.tie(0);//这条比较重要

//cout.tie(0);//这句貌似似乎可以不要

ios::sync\_with\_stdio(false);

## 模板代码头

#include <iostream>

#include <iomanip>

#include <fstream>

#include <sstream>

#include <cmath>

#include <cstdio>

#include <cstring>

#include <cctype>

#include <algorithm>

#include <functional>

#include <numeric>

#include <string>

#include <set>

#include <map>

#include <stack>

#include <vector>

#include <queue>

#include <deque>

#include <list>

using namespace std;

const int INF=0x7fffffff;//7+7\*f

int main(void)

{

return 0;

}

//<bits/stdc++.h> 包括极大部分头文件（国外OJ可用）

## 格式化输入输出:

### 忽略变量（%\*?）

// 例

int i;

scanf("%\*d %d", &i); // input: 1 2

printf("%d", i); // output: 2

用途1：控制扫描位数

// 例 读取身份证号中的年月日

int y, m, d;

scanf("%\*6d%4d%2d%2d%\*s", &y, &m, &d); // input: 44142119940704057X

printf("%d %d %d", y, m, d); // output: 1994 7 4

### 扫描集（%[]）

//说明：用于字符串读取

用法1：scanf("%[0-9]",a) //只读取数字

input:123a456 output:123

用法2: scanf("%[13579]",a)

input:123 output : 1

用法3: scanf("%[^0-9]",a) //只读取非数字字符

input: abc00qfdf output : abc

### 输出前面带有0

说明：%02d //整数不够2位用0填补空处

用途1: 输出xxxx-xx-xx的日期格式

int y = 2014;

int m = 7;

int d = 5;

printf("%04d-%02d-%02d", y, m, d); // output: 2014-07-05

### %n

说明：将本次scanf()调用到%n之前所读取的字符数量存储对应变量中

用途1：scanf("%d%n",&a,&n) input 101

printf("%d",n) output 3

### getline() 用法

头文件为<string>

基本用法：

string str;

getline(cin, str);

getline(cin, str, '\n');

默认结束符为回车（\n），结束符可自己设定

无论是开头或是结尾都不会有跳过行为，以结束符为标志结束读入

输入结束时会将结束符也一起读入，并将其丢弃，读取完毕后结束符不会留在缓冲区（输入队列）中

### C++输出格式

|  |  |  |  |
| --- | --- | --- | --- |
| 操纵符 | 含义 | 操纵符 | 含义 |
| dec/oct/hex | 十/八/十六进制格式 | showpoint/ noshowpoint | 是否输出小数点和尾部0 |
| setbase(n) | 转成n进制，n只能为8、10、16 | showpos/noshowpos | 是否正数前输出“+” |
| showbase/noshowbase | 是否显示0x和0 |  |  |
| uppercase/nouppercase | 显示0X还是0x | setprecision(n) | 设置浮点数精度 |
|  |  | cout.precision() | 返回精度 |
| setw(n) | 设置输出宽度为n（只作用1次） | fixed | 绝对位数小数形式 |
| left | 左对齐 | scientific | 指数形式 |
| right | 右对齐 | cout.unsetf(ostream::floatfield) | 恢复默认小数输出格式 |
| setfill(ch) | 设置填充字符ch |  |  |

## algorithm 的常用函数

### fill

说明：用100填充a数组

基本用法：fill(a,a+n,100)

### next\_permutation

说明：取出当前范围内的排列，并将其重新排序为下一个排列

基本用法：

int a[5] = {1,2,3,4,5};

do

{

for(int i = 0;i < 5;i++)

cout << a[i] << endl;

} while (next\_permutation(a,a+5));

//若没有下个排列会返回false，并将数组置为第一个序列

// prev\_permutation逆序(越变越小)

### reverse

说明：将范围内元素重新按反序排列。

### sort

说明：以升序重新排列范围内的元素

基本用法：

bool comp(pair<long long,long long> a,pair<long long,long long> b)

{

return a.first<b.first;

}

### max\_element / min\_element

说明：函数找最大 / 小元素 （区间前闭后开）

基本用法：

min\_element(name,name+n)-name // min\_element返回的是一个数组的下标的指针,上述是返回最小值的下标

如果要输出数组的最小值，之后却要对原数组进行操作，则可以用min\_element(n,n+len),max\_element(n,n+len)//保证了原数组的完整

### map

说明：遍历

map<string,int>::iterator ite;

for(ite = team.begin();ite != team.end();ite++)

if(ite->second == max)

cout << ite->first << endl;

反向遍历迭代器要定义为reverse\_iterator，然后等于name.rbegin()

### lower\_bound() ,upper\_bound()

前提：已排序

说明：lower\_bound 查找“大于或者等于x的第一个位置”

upper\_bound 返回"最后一个大于或等于x的位置"

基本用法：

第一个参数.begin()，第二个参数.end(),第三个参数为要查找的值，返回的值是一个迭代器，要知道第几位需要减去首地址。 int pos = lower\_bound(a,a + n,x) - a; //在已排序数组a中寻找x

### binary\_search (a.begin(),a,end(),num)

说明：判断某区间内是否包含某个元素

### includes() 判断某区间内是否都涵盖于另一区间中

Includes(a.begin,a.end(),b.begin(),b.end()) a,b需要排序好

## 其他

### 避免对浮点数进行 == 或 ！= 比较

解决方法：#define EPSILON 1e6

if (fabs(x - y) <= EPSILON)

### 取非负数

p = max(0,num);

### 各类型长度

说明：

1.int和long的最大值为2^31-1，即2147483648(共10位)，取值范围为[-2147483648,2147483648];

2.long long的最大值为2^63-1，即9223372036854775807(共19位),取值范围为[-9223372036854775807,9223372036854775807];

3.double的最大值为1.79769e+308，最小正值为2.22507e-308。

### 随机数

#define random(a,b) ((a)+rand()%((b)-(a)+1))

# 数学概念

## PI精确赋值

PI = acos(-1.0)

## cmath常用函数

说明：

1.C语言头文件是<math.h>，C++是<cmath>；

2.注意ceil()和floor()返回类型是double，

使用 cout << ceil(x); 输出时不会带小数但是当数字大时会使用科学记数法，

因此使用时最好写成 cout << (int)ceil(x);

1.log()

基本用法：

log(x); // x可为int、double、float、long double类型，返回值为double、float、long double类型

log(x)/log(10); // 换底公式，所求结果为10为底的对数

log10() //以10为底的对数

2.取整函数

ceil() // 含义：向上取整

floor() // 含义：向下取整

## 最大公约数

qgcd&&lcm-最大公约数与最小公倍数

int qgcd(inta, intb){

if(a == 0) returnb;

if(b == 0) returna;

if(!(a& 1) && !(b& 1)) return kgcd(a>>1, b>>1) << 1;

elseif(!(b& 1)) return kgcd(a, b>>1);

elseif(!(a& 1)) return kgcd(a>>1, b);

elsereturn kgcd(abs(a - b), min(a, b));

}

int lcm(inta, intb) {

returna/qgcd(a, b)\*b;

}

// 附上辗转相除法的gcd

int gcd(inta, intb) {

returnb==0 ? a : gcd(b, a%b);

}

## 快速幂

#define LL longlong

LL qpow(LL a, LL n，LL mod) {

LL ret = 1;

while (n) {

if (n & 1) ret = ret \* a % mod;

a = a \* a % mod;

n>>= 1;

}

return ret;

}

## isPrime-判断素数

/\* isPrime-判断素数 \*/

/\*

bool isPrime(int n) {

if (n<= 1) return false;

else if (n == 2) return true;

else if (n % 2 == 0) false;

else

{

double limit = sqrt(n) ;

for (int i = 3; i <= limit; i += 2)

{

if (n % i == 0) return false;

}

return true;

}

}

## 筛选法求素数（）

int isp[MAX];

void intial()

{

for(int i=0;i<MAX;i++)

{

isp[i]=1;

}

for(int i=2;i<MAX;i++)

{

int it=2;

if(!isp[i])continue;

while(i\*it<MAX)

{

isp[i\*it]=0;

it++;

}

}

isp[1]=isp[0]=0;

}

## 组合函数C(n, r)

### 基本用法

int com(int n, int r){// return C(n, r)

if( n-r < r ) r = n-r; // C(n, r) = C(n, n-r)

int i, j, s = 1;

for( i=0, j=1; i <r; ++i ){

s \*= (n-i);

for( ; j <= r&& s%j == 0; ++j ) s /= j;

}

return s;

}

### 性质

前提：端点的数为1.

每个数等于它上方两数之和。

每行数字左右对称，由1开始逐渐变大。

第n行的数字有n项。

第n行数字和为2n-1。

第n行的m个数可表示为 C(n-1,m-1)，即为从n-1个不同元素中取m-1个元素的组合数。

第n行的第m个数和第n-m+1个数相等 ，为组合数性质之一。

每个数字等于上一行的左右两个数字之和。可用此性质写出整个杨辉三角。即第n+1行的第i个数等于第n行的第i-1个数和第i个数之和，这也是组合数的性质之一。即 C(n+1,i)=C(n,i)+C(n,i-1)。

(a+b)n的展开式中的各项系数依次对应杨辉三角的第(n+1)行中的每一项。

将第2n+1行第1个数，跟第2n+2行第3个数、第2n+3行第5个数……连成一线，这些数的和是第4n+1个斐波那契数；将第2n行第2个数(n>1)，跟第2n-1行第4个数、第2n-2行第6个数……这些数之和是第4n-2个斐波那契数。

将各行数字相排列，可得11的n-1（n为行数）次方：1=11^0; 11=11^1; 121=11^2……当n≥5时会不符合这一条性质，此时应把第n行的最右面的数字"1"放在个位，然后把左面的一个数字的个位对齐到十位... ...，以此类推，把空位用“0”补齐，然后把所有的数加起来，得到的数正好是11的n-1次方。以n=11为例，第十一行的数为：1,10,45,120,210,252,210,120,45,10,1，结果为 25937424601=1110。

判断 C(n,r)是奇数还是偶数

公式： C(n,r) 为奇数时 n&r== r

### 应用

由1开始，正整数在杨辉三角形出现的次数为∞,1, 2, 2, 2, 3, 2, 2, 2, 4, 2, 2, 2, 2, 4, ... （OEIS:A003016）。最小而又大于1的数在贾宪三角形至少出现n次的数为2, 3, 6, 10, 120, 120, 3003, 3003, ... （OEIS:A062527）

除了1之外，所有正整数都出现有限次，只有2出现刚好一次，6,20,70等出现三次；出现两次和四次的数很多，还未能找到出现刚好五次的数。120,210,1540等出现刚好六次。（OEIS:A098565）

因为丢番图方程 有无穷个解，所以出现至少六次的数有无穷个多。解为，其中Fn表示第n个斐波那契数（F1=F2=1）。3003是第一个出现八次的数。

### lucas定理

Lucas 定理：A、B是非负整数，p是质数。AB写成p进制：A=a[n]a[n-1]...a[0]，B=b[n]b[n-1]...b[0]。

则组合数C(A,B)与C(a[n],b[n])\*C(a[n-1],b[n-1])\*...\*C(a[0],b[0]) modp同

即：Lucas(n,m,p)=c(n%p,m%p)\*Lucas(n/p,m/p,p)

#include <iostream>

#include <cstdio>

#include <algorithm>

#include <cmath>

#include <cstring>

using namespace std;

typedef long long lld;

lld n, m, p;

lld Ext\_gcd(lld a,lld b,lld &x,lld &y){

if(b==0) { x=1, y=0; return a; }

lld ret= Ext\_gcd(b,a%b,y,x);

y-= a/b\*x;

return ret;

}

lld Inv(lld a,int m){ ///求逆元

lld d,x,y,t= (lld)m;

d= Ext\_gcd(a,t,x,y);

if(d==1) return (x%t+t)%t;

return -1;

}

lld Cm(lld n, lld m, lld p) ///组合数学

{

lld a=1, b=1;

if(m>n) return 0;

while(m)

{

a=(a\*n)%p;

b=(b\*m)%p;

m--;

n--;

}

return (lld)a\*Inv(b,p)%p; ///（a/b）%p 等价于 a\*（b，p）的逆元

}

int Lucas(lld n, lld m, lld p) ///把n分段递归求解相乘

{

if(m==0) return 1;

return (lld)Cm(n%p,m%p,p)\*(lld)Lucas(n/p,m/p,p)%p;

}

int main()

{

int T;

cin >> T;

while(T--)

{

scanf("%lld%lld%lld",&n,&m,&p);

printf("%d\n",Lucas(n+m,m,p));

}

return 0;

}

## 斐波那契数列性质

2.8.1平方与前后项

从第二项开始，每个奇数项的平方都比前后两项之积少1，每个偶数项的平方都比前后两项之积多1。

如：第二项1的平方比它的前一项1和它的后一项2的积2少1，第三项2的平方比它的前一项1和它的后一项3的积3多1。

（注：奇数项和偶数项是指项数的奇偶，而并不是指数列的数字本身的奇偶，比如从数列第二项1开始数，第4项5是奇数，但它是偶数项，如果认为5是奇数项，那就误解题意，怎么都说不通）

证明经计算可得：[f(n)]^2-f(n-1)f(n+1)=(-1)^(n-1)

2.8.2与集合子集

斐波那契数列的第n+2项同时也代表了集合{1,2,...,n}中所有不包含相邻正整数的子集个数。

2.8.3奇数项求和：http://a.hiphotos.baidu.com/baike/s%3D214/sign=eb8d4138f536afc30a0c38648718eb85/1c950a7b02087bf4e2135b11f0d3572c10dfcf4f.jpg

2.8.4偶数项求和：http://h.hiphotos.baidu.com/baike/s%3D247/sign=53440a65c9ef7609380b9e9b19dca301/9825bc315c6034a8016fb679c9134954082376bc.jpg

2.8.5平方求和：http://g.hiphotos.baidu.com/baike/s%3D227/sign=5ead1ee5d62a283447a631096cb7c92e/a2cc7cd98d1001e907f51af7ba0e7bec55e79778.jpg

2.8.6隔项关系 f(2n-2m-2)[f(2n)+f(2n+2)]=f(2m+2)+f(4n-2m) [ n〉m≥-1，且n≥1]

2.8.7两倍项关系 f(2n)/f(n)=f(n-1)+ f(n+1)

## 约瑟夫环

int f[1000000]={0};

f[1]=0;

for(int i=2 ; i<=n ; ++i)

if(!f[i])f[i]=(f[i-1]+7)%i;

cout<<f[n]+1<<"\n";//从0开始计数则不用加1

## 阶乘最后非0位

说明：

1.该算法复杂度为O(N\*logN)。

//返回该位, n以字符串方式传入

#include<string.h>

#defineMAXN 10000

int lastdigit(char\* buf) {

constint mod[20]={1,1,2,6,4,2,2,4,2,8,4,4,8,4,6,8,8,6,8,2};

int len=strlen(buf),a[MAXN],i,c,ret=1;

if (len==1) return mod[buf[0]-'0'];

// 下面的for是把字符串n变成数组表示，且为倒序

for (i=0;i<len;i++) a[i]=buf[len-1-i]-'0';

for (;len;len-=!a[len-1]){

ret=ret\*mod[a[1]%2\*10+a[0]]%10; // 此操作是对20取模

// 下一个for是对n进行除5运算

for (c=0,i=len-1;i>=0;i--)

c=c\*10+a[i],a[i]=c/5,c%=5;

}

return ret;

}

## 数学公式

### 勾股数

设a,b,c是直角三角形的三条边，a,b是直角边，c是斜边，则有

a = 2\*m\*n

b = m^2 - n^2

c = m^2 + n^2

该公式只能求出基本勾股数，无法求出派生勾股数，如6,8,10无法求出来

还需要注意两点

1.由于两边之和要大于第三边，所以有

a\*m\*n + m^2-n^2 > m^2+n^2

即m > n

2.a,b,c必须互质，否则无法得到派生的勾股数

### n条直线最多划分成n\*(n+1)/2 + 1个平面

### 求和公式

k = 1..n

1. sum( k ) = n(n+1)/2

2. sum( 2k-1 ) = n^2

3. sum( k^2 ) = n(n+1)(2n+1)/6

4. sum( (2k-1)^2 ) = n(4n^2-1)/3

5. sum( k^3 ) = (n(n+1)/2)^2

6. sum( (2k-1)^3 ) = n^2(2n^2-1)

7. sum( k^4 ) = n(n+1)(2n+1)(3n^2+3n-1)/30

8. sum( k^5 ) = n^2(n+1)^2(2n^2+2n-1)/12

9. sum( k(k+1) ) = n(n+1)(n+2)/3

10. sum( k(k+1)(k+2) ) = n(n+1)(n+2)(n+3)/4

11. sum( k(k+1)(k+2)(k+3) ) = n(n+1)(n+2)(n+3)(n+4)/5

### Sn=1+1/2+1/3+…+1/n

= ln(n)+ γ

欧拉函数γ= 0.57721566490153286060651209

### 判断n!是否能够被m整除

计算方法是把m进行质因数分解，看下m的每一个质因数是否能够在n!中找到，如果有一个没有被包含，那么就不能被整除！

推广：n!中间包含了多少个x(x是任意的一个数，不过一般情况下我们都只讨论x为质数)

答： n/x+n/(x^2)+n/(x^3).....[一直加到x的乘方不超过n]

### 因子和的计算方法

计算方法是把12分解为质因数的表达形式2^2\*3

那么他的因子和就是：(1+2+2^2)\*(1+3)

### 哥德巴赫猜想

一个大偶数(>=4)必然可以拆分为两个素数的和，

推广: 任意一个>=8的整数一定能够拆分为四个素数的和

### 一绳长L，一蚂蚁从绳的一段爬向另一端，速度为v m/s，同时绳子以每秒u米的速度均匀伸长，问： 蚂蚁能否到达绳的另一端？如能，需要多长时间？

公式：T = [pow(e,u/v) - 1] \* L / u

## 矩阵乘法

const int N = 15486042;

int MOD=9342001;//就是结果求余

#define LL long long

long long now,nxt;

struct Matrix

{

LL ma[2][2];

Matrix()

{

ma[0][0]=ma[1][1]=1;

ma[0][1]=ma[1][0]=0;

}

//下面放的是你推关系自己写出来的矩阵

void init()

{

ma[0][0]=4;

ma[1][0]=1;

ma[0][1]=-1;

ma[1][1]=0;

}

//放自己写的矩阵规则

Matrix operator \* (const Matrix &a) const

{

Matrix res;

for ( int i = 0 ; i < 2 ; i++ )

for ( int j = 0 ; j < 2 ; j++ )

{

res.ma[i][j]=0;

for ( int k = 0 ; k < 2 ; k++ )

res.ma[i][j]=(res.ma[i][j]+ma[i][k]\*a.ma[k][j]+MOD)%MOD;

res.ma[i][j]%=MOD;

}

return res;

}

};

void init(int k)

{

Matrix res, a;

a.init();

while (k)

{

if (k&1)

res = res\*a;

a=a\*a;

k>>=1;

}

now= (res.ma[1][0])%MOD;//返回结果。

// nxt =(res.ma[0][0]+res.ma[1][0])%MOD;是这个规律的下一个值。

## 模的性质

### 基本性质

若p|(a-b)，则a≡b (% p)。例如 11 ≡ 4 (% 7)， 18 ≡ 4(% 7)

(a % p)=(b % p)意味a≡b (% p)

对称性：a≡b (% p)等价于b≡a (% p)

传递性：若a≡b (% p)且b≡c (% p) ，则a≡c (% p)

### 运算规则

模运算与基本四则运算有些相似，但是除法例外。其规则如下：

a ^ b % p = ((a % p)^b) % p

### 重要定理

若a≡b (% p)，则对于任意的c，都有(a + c) ≡ (b + c) (%p)；

若a≡b (% p)，则对于任意的c，都有(a \* c) ≡ (b \* c) (%p)；

若a≡b (% p)，c≡d (% p)，则 (a + c) ≡ (b + d) (%p)，(a - c) ≡ (b - d) (%p)， (a \* c) ≡ (b \* d) (%p)，(a / c) ≡ (b / d) (%p)；

## 数字取整性质

若一个整数的末位是0、2、4、6或8，则这个数能被2整除。

若一个整数的数字和能被3整除，则这个整数能被3整除。

能被4整除的数，个位和十位所组成的两位数能被4整除，那么这个数能被4整除

若一个整数的末位是0或5，则这个数能被5整除。

能被6整除的数，各数位上的数字和能被3整除的偶数，如果一个数既能被2整除又能被3整除，那么这个数能被6整除

能被7整除的数，若一个整数的个位数字截去，再从余下的数中，减去个位数的2倍，如果差是7的倍数，则原数能被7整除。如果差太大或心算不易看出是否7的倍数，就需要继续上述「截尾、倍大、相减、验差」的过程，直到能清楚判断为止。例如，判断133是否 7的倍数的过程如下：13－3×2＝7，所以133是7的倍数；又例如判断6139是否7的倍数的过程如下：613－9×2＝595 ， 59－5×2＝49，所以6139是7的倍数，余类推。

能被8整除的数，一个整数的末3位若能被8整除，则该数一定能被8整除。

能被9整除的数，各个数位上的数字和能被9整除，那么这个数能被9整除

能被13整除的数，若一个整数的个位数字截去，再从余下的数中，加上个位数的4倍，如果差是13的倍数，则原数能被13整除。如果差太大或心算不易看出是否13的倍数，就需要继续上述「截尾、倍大、相加、验差」的过程，直到能清楚判断为止。

能被17整除的数，若一个整数的个位数字截去，再从余下的数中，减去个位数的5倍，如果差是17的倍数，则原数能被17整除。如果差太大或心算不易看出是否17的倍数，就需要继续上述「截尾、倍大、相减、验差」的过程，直到能清楚判断为止。另一种方法：若一个整数的末三位与3倍的前面的隔出数的差能被17整除，则这个数能被17整除

能被19整除的数，若一个整数的个位数字截去，再从余下的数中，加上个位数的2倍，如果差是19的倍数，则原数能被19整除。如果差太大或心算不易看出是否19的倍数，就需要继续上述「截尾、倍大、相加、验差」的过程，直到能清楚判断为止。另一种方法：若一个整数的末三位与7倍的前面的隔出数的差能被19整除，则这个数能被19整除

能被23整除的数，若一个整数的末四位与前面5倍的隔出数的差能被23(或29)整除，则这个数能被23整除

能被25整除的数，十位和个位所组成的两位数能被25整除。

能被125整除的数，百位、十位和个位所组成的三位数能被125整除。

## 欧拉函数

#include <iostream>

#include <cstdio>

#include <cmath>

using namespace std;

#define bint \_\_int64

#define N 3000001

bint phi[N];

void init()

{

int i, j;

for(i = 1; i < N; i++)

phi[i] = i;

for(i = 2; i < N; i++)

if(i == phi[i]) //此时i为素数

for(j = i; j < N; j += i) //j累加i

phi[j] = (phi[j] / i) \* (i - 1); //j有因子i,而且i是素数,正是欧拉函数

}

int main()

{

init();

int a, b;

while(scanf("%d%d", &a, &b) != EOF)

{

bint ans = 0;

for(int i = a; i <= b; i++)

ans += phi[i];

printf("%I64d\n", ans);

}

return 0;

}

## 三分法求函数极值

//好像只适用于只有一个峰的函数

double f(double tan ,double H, double h, double D)//自定函数

{

return H+D-(D\*tan+(H-h)/tan);

}

int main()

{

int T;

scanf("%d",&T);

while(T--)

{

double H,h,D;

scanf("%lf%lf%lf",&H,&h,&D);

double left=(H-h)/D,right=H/D;//初始左右范围

int size=100;//计算次数

while(size--)

{

double mid=(left+right)/2;

double midmid=(mid+right)/2;

if(f(mid,H,h,D)>f(midmid,H,h,D))//调用

right=midmid;

else

left=mid;

}

printf("%.3f\n",f(left,H,h,D));//调用

}

return 0;

}

# 位运算

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 位运算 | 示例 | 功能 | 位运算 | 示例 | 功能 |
| X >> 1 | 101101->10110 | 去掉最后一位 | X << 1 | 101101->1011010 | 在最后加一个0 |
| x ^ 1 | 101101->101100 | 最后一位取反 | x << 1+1 | 101101->1011011 | 在最后加一个1 |
| x | 1 | 101100->101101 | 把最后一位变成1 | x | 1-1 | 101101->101100 | 把最后一位变成0 |
| x & ((1 < < k)-1) | 1101101->1101,k=5 | 取末k位 | x | (1 < < k-1) | 101001->101111,k=4 | 把末k位变成1 |
| x&((1<<k)-1) |  | x mod(2^k) | x | (x-1) | 11011000->11011111 | 把右边连续的0变成1 |
| !(x&(x-1)))&&x |  | 判断x是否为2的正整数幂 （去掉&&x则为非负整数幂） | x | (x+1) | 100101111->100111111 | 把右起第一个0变成1 |
| (x&1)==1 |  | 判断奇数 | (x&1)==0 |  | 判断偶数 |

# 日期

## 判断是否为闰年

bool IsLeap(int year)

{

return (year % 4 ==0 || year % 400 ==0) && (year % 100 !=0);

}

## 蔡勒公式

http://a.hiphotos.baidu.com/baike/s%3D301/sign=2bb7d34afe039245a5b5e70fb695a4a8/f636afc379310a555b60fb73b54543a9832610f4.jpghttp://d.hiphotos.baidu.com/baike/s%3D377/sign=1dc66b3e9c510fb37c197190ee32c893/b151f8198618367a21bf5a5a2c738bd4b31ce507.jpg说明：计算某日期是星期几

公式：

若要计算的日期是在1582年10月4日或之前，公式则为

符号意义：w：星期； w对7取模得：0-星期日~ 6-星期六

c：世纪减1（年份前两位数）

y：年（后两位数）

m：月（m大于等于3，小于等于14，即在蔡勒公式中，某年的1、2月要看作上一年的13、14月来计算)

d：日

[ ]代表取整，即只要整数部分。

代码：

int main(){

int year, month, day;

while(cin >> year >> month >> day){

if(month < 3）{

year -= 1;

month += 12;

}

char b[7][10]={"sunday","monday","tuesday","wednesday","thursday","friday","saturday"};

int c = int (year / 100）, y = year - 100 \* c;

int w = int ( c / 4) - 2 \* c + y + int ( y / 4 ) +(26 \* ( month + 1 ) / 10) + day - 1;

w = (w % 7 + 7) % 7;

cout << b[w] << endl;

## 计算日期相差天数

说明：

1.此算法由yybird编写；

2.此处将相隔天数当作两天来算；

3.该算法只限用于公元前的计算。

#include<iostream>

#include<cstdio>

#defineSWAP(x, y) { x = x + y; y = x - y; x = x - y; }

#defineISLEAP(x) x%100!=0 && x%4==0 || x%400==0?1:0

usingnamespace std;

int month[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

int numOfLeapYear(inty) {

int num = 0;

for (int i = 4; i <= y; i+=4) {

if (i % 100 != 0 || i % 400 == 0)

num++;

}

return num;

}

int main() {

int y1, y2, m1, m2, d1, d2;

while (scanf("%4d%2d%2d", &y1, &m1, &d1) != EOF) {

scanf("%4d%2d%2d", &y2, &m2, &d2);

int com1 = y1 \* 10000 + m1 \* 100 + d1;

int com2 = y2 \* 10000 + m2 \* 100 + d2;

if (com2 < com1) {

SWAP(y1, y2);

SWAP(m1, m2);

SWAP(d1, d2);

}

int sum1 = (y1-1) \* 365 + numOfLeapYear(y1-1) + d1;

int sum2 = (y2-1) \* 365 + numOfLeapYear(y2-1) + d2;

for (int i = 0; i < m1-1; i++) {

if (i == 1 &&ISLEAP(y1))

sum1++;

sum1 += month[i];

}

for (int i = 0; i < m2-1; i++) {

if (i == 1 &&ISLEAP(y2))

sum2++;

sum2 += month[i];

}

cout << sum2 - sum1 + 1 << endl; // 若相隔天数只当一天，则去掉+1

}

return 0;

}

## 计算日期相差天数（封装成函数）

说明：

该算法只限用于公元前的计算。

#define ISLEAP(x) （x % 100 != 0 && x % 4 == 0 || x % 400 == 0 ? 1 : 0）

const int s[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

// 从公元0年开始到那天为止所经历的闰年的数量

int leap(inty) {

if (!y) return 0;

returny/4 - y/100 + y/400;

}

int calc(intd, intm, inty) {

int res = (y-1) \* 365 + leap(y-1);

for (int i = 1; i <m; i++)

res+= s[i];

if (ISLEAP(y) && m > 2) res++;

res += d;

return res;

}

int countDay(intd1, intm1, inty1, intd2, intm2, inty2) {

int res1 = calc(d1, m1, y1);

int res2 = calc(d2, m2, y2);

if (res1 > res2) return res1 - res2;

else return res2 - res1;

}

# 字符串相关

## string常用库函数

### string类的构造函数

string(const char \*s); //用c字符串s初始化

string(int n,char c); //用n个字符c初始化

此外，string类还支持默认构造函数和复制构造函数，如string s1；string s2="hello"；都是正确的写法。当构造的string太长而无法表达时会抛出length\_error异

### string类的字符操作

const char &operator[](int n)const;

const char &at(int n)const;

char &operator[](int n);

char &at(int n);

operator[]和at()均返回当前字符串中第n个字符的位置，但at函数提供范围检查，当越界时会抛出out\_of\_range异常，下标运算符[]不提供检查访问。

const char \*data()const;//返回一个非null终止的c字符数组

const char \*c\_str()const;//返回一个以null终止的c字符串

int copy(char \*s, int n, int pos = 0) const;//把当前串中以pos开始的n个字符拷贝到以s为起始位置的字符数组中，返回实际拷贝的数目

### string的特性描述

int capacity()const; //返回当前容量（即string中不必增加内存即可存放的元素个数）

int max\_size()const; //返回string对象中可存放的最大字符串的长度

int size()const; //返回当前字符串的大小

int length()const; //返回当前字符串的长度

bool empty()const; //当前字符串是否为空

void resize(int len,char c);//把字符串当前大小置为len，并用字符c填充不足的部分

### string类的输入输出操作

string类重载运算符operator>>用于输入，同样重载运算符operator<<用于输出操作。

函数getline(istream &in,string &s);用于从输入流in中读取字符串到s中，以换行符'\n'分开。

### string的赋值

string &operator=(const string &s);//把字符串s赋给当前字符串

string &assign(const char \*s);//用c类型字符串s赋值

string &assign(const char \*s,int n);//用c字符串s开始的n个字符赋值

string &assign(const string &s);//把字符串s赋给当前字符串

string &assign(int n,char c);//用n个字符c赋值给当前字符串

string &assign(const string &s,int start,int n);//把字符串s中从start开始的n个字符赋给当前字符串

string &assign(const\_iterator first,const\_itertor last);//把first和last迭代器之间的部分赋给字符串

### string的连接

string &operator+=(const string &s);//把字符串s连接到当前字符串的结尾

string &append(const char \*s); //把c类型字符串s连接到当前字符串结尾

string &append(const char \*s,int n);//把c类型字符串s的前n个字符连接到当前字符串结尾

string &append(const string &s); //同operator+=()

string &append(const string &s,int pos,int n);//把字符串s中从pos开始的n个字符连接到当前字符串的结尾

string &append(int n,char c); //在当前字符串结尾添加n个字符c

string &append(const\_iterator first,const\_iterator last);//把迭代器first和last之间的部分连接到当前字符串的结尾

### string的比较

bool operator==(const string &s1,const string &s2)const;//比较两个字符串是否相等

运算符">","<",">=","<=","!="均被重载用于字符串的比较；

int compare(const string &s) const;//比较当前字符串和s的大小

int compare(int pos, int n,const string &s)const;//比较当前字符串从pos开始的n个字符组成的字符串与s的大小

int compare(int pos, int n,const string &s,int pos2,int n2)const;//比较当前字符串从pos开始的n个字符组成的字符串与s中pos2开始的n2个字符组成的字符串的大小

int compare(const char \*s) const;

int compare(int pos, int n,const char \*s) const;

int compare(int pos, int n,const char \*s, int pos2) const;

compare函数在>时返回1，<时返回-1，==时返回0

### string的子串

string substr(int pos = 0,int n = npos) const;//返回pos开始的n个字符组成的字符串

### string的交换

void swap(string &s2); //交换当前字符串与s2的值

### string类的查找函数

int find(char c, int pos = 0) const;//从pos开始查找字符c在当前字符串的位置

int find(const char \*s, int pos = 0) const;//从pos开始查找字符串s在当前串中的位置

int find(const char \*s, int pos, int n) const;//从pos开始查找字符串s中前n个字符在当前串中的位置

int find(const string &s, int pos = 0) const;//从pos开始查找字符串s在当前串中的位置

//查找成功时返回所在位置，失败返回string::npos的值

int rfind(char c, int pos = npos) const;//从pos开始从后向前查找字符c在当前串中的位置

int rfind(const char \*s, int pos = npos) const;

int rfind(const char \*s, int pos, int n = npos) const;

int rfind(const string &s,int pos = npos) const;

//从pos开始从后向前查找字符串s中前n个字符组成的字符串在当前串中的位置，成功返回所在位置，失败时返回string::npos的值

int find\_first\_of(char c, int pos = 0) const;//从pos开始查找字符c第一次出现的位置

int find\_first\_of(const char \*s, int pos = 0) const;

int find\_first\_of(const char \*s, int pos, int n) const;

int find\_first\_of(const string &s,int pos = 0) const;

//从pos开始查找当前串中第一个在s的前n个字符组成的数组里的字符的位置。查找失败返回string::npos

int find\_first\_not\_of(char c, int pos = 0) const;

int find\_first\_not\_of(const char \*s, int pos = 0) const;

int find\_first\_not\_of(const char \*s, int pos,int n) const;

int find\_first\_not\_of(const string &s,int pos = 0) const;

//从当前串中查找第一个不在串s中的字符出现的位置，失败返回string::npos

int find\_last\_of(char c, int pos = npos) const;

int find\_last\_of(const char \*s, int pos = npos) const;

int find\_last\_of(const char \*s, int pos, int n = npos) const;

int find\_last\_of(const string &s,int pos = npos) const;

int find\_last\_not\_of(char c, int pos = npos) const;

int find\_last\_not\_of(const char \*s, int pos = npos) const;

int find\_last\_not\_of(const char \*s, int pos, int n) const;

int find\_last\_not\_of(const string &s,int pos = npos) const;

//find\_last\_of和find\_last\_not\_of与find\_first\_of和find\_first\_not\_of相似，只不过是从后向前查找

### string类的替换函数

string &replace(int p0, int n0,const char \*s);//删除从p0开始的n0个字符，然后在p0处插入串s

string &replace(int p0, int n0,const char \*s, int n);//删除p0开始的n0个字符，然后在p0处插入字符串s的前n个字符

string &replace(int p0, int n0,const string &s);//删除从p0开始的n0个字符，然后在p0处插入串s

string &replace(int p0, int n0,const string &s, int pos, int n);//删除p0开始的n0个字符，然后在p0处插入串s中从pos开始的n个字符

string &replace(int p0, int n0,int n, char c);//删除p0开始的n0个字符，然后在p0处插入n个字符c

string &replace(iterator first0, iterator last0,const char \*s);//把[first0，last0）之间的部分替换为字符串s

string &replace(iterator first0, iterator last0,const char \*s, int n);//把[first0，last0）之间的部分替换为s的前n个字符

string &replace(iterator first0, iterator last0,const string &s);//把[first0，last0）之间的部分替换为串s

string &replace(iterator first0, iterator last0,int n, char c);//把[first0，last0）之间的部分替换为n个字符c

string &replace(iterator first0, iterator last0,const\_iterator first, const\_iterator last);//把[first0，last0）之间的部分替换成[first，last）之间的字符串

### string类的插入函数

string &insert(int p0, const char \*s);

string &insert(int p0, const char \*s, int n);

string &insert(int p0,const string &s);

string &insert(int p0,const string &s, int pos, int n);

//前4个函数在p0位置插入字符串s中pos开始的前n个字符

string &insert(int p0, int n, char c);//此函数在p0处插入n个字符c

iterator insert(iterator it, char c);//在it处插入字符c，返回插入后迭代器的位置

void insert(iterator it, const\_iterator first, const\_iterator last);//在it处插入[first，last）之间的字符

void insert(iterator it, int n, char c);//在it处插入n个字符c

### string类的删除函数

iterator erase(iterator first, iterator last);//删除[first，last）之间的所有字符，返回删除后迭代器的位置

iterator erase(iterator it);//删除it指向的字符，返回删除后迭代器的位置

string &erase(int pos = 0, int n = npos);//删除pos开始的n个字符，返回修改后的字符串

### string类的迭代器处理

string类提供了向前和向后遍历的迭代器iterator，迭代器提供了访问各个字符的语法，类似于指针操作，迭代器不检查范围。

用string::iterator或string::const\_iterator声明迭代器变量，const\_iterator不允许改变迭代的内容。常用迭代器函数有：

const\_iterator begin()const;

iterator begin(); //返回string的起始位置

const\_iterator end()const;

iterator end(); //返回string的最后一个字符后面的位置

const\_iterator rbegin()const;

iterator rbegin(); //返回string的最后一个字符的位置

const\_iterator rend()const;

iterator rend(); //返回string第一个字符位置的前面

rbegin和rend用于从后向前的迭代访问，通过设置迭代器string::reverse\_iterator,string::const\_reverse\_iterator实现

# 高精度算法与进制转换

## 十进制转m进制

说明：

1.该算法由yybird编写；

1.随着所转换的数值的大小的变化，应当修改stack的大小；

2.n是所需转换的数,m是需要转换成的进制数。

void transfer(intn, intm) {

int top = -1, stack[16]; // 16：可用于计算一个不超过32768的数

for (; n || top == -1; n /= m) // 注：当输入的n本身就为0时top一定为-1，此时循环会执行一次并往栈中放入0，然后停止

stack[++top] = n % m;

while (top >= 0) // 将数字输出，top到0时会在printf里面再减一次，达到空栈时的-1

printf("%d", stack[top--]);

printf("\n");

}

## N进制转换十进制

第i位上的数字乘以要转换的进制的i次方

## 长数字相乘

说明：

1.getdigits是将字符串换成每一位数字保存到整型数组中；

2.根据需要修改N的值，并注意数组c的大小为2\*N。

#include<iostream>

#include<string>

usingnamespace std;

#define N 100

/\*

\*将在数组中保存的字符串转成数字存到int数组中

\*/

void getdigits(int \*a, string s) {

char digit;

int len = s.length();

//对数组初始化

for (int i = 0; i < N; ++i)

a[i] = 0;

for (int i = 0; i < len; ++i){

digit = s[i];

a[len - 1 - i] = digit - '0';//字符串s="12345",因此第一个字符应该存在int数组的最后一个位置

}

}

/\*

\*将数组a与数组b逐位相乘以后存入数组c

\*/

void multiply(int \*a, int \*b, int \*c) {

int i, j;

//数组初始化

for (i = 0; i < 2 \* N; ++i)

c[i] = 0;

/\*

\*数组a中的每位逐位与数组b相乘，并把结果存入数组c

\*比如，12345\*12345，a中的5与12345逐位相乘

\*对于数组c：\*(c+i+j)在i=0,j=0,1,2,3,4时存的是5与各位相乘的结果

\*而在i=1,j=0,1,2,3,4时不光会存4与各位相乘的结果，还会累加上上次相乘的结果.这一点是需要注意的!!!

\*/

for (i = 0; i < N; ++i)

for (j = 0; j < N; ++j)

c[i + j] += a[i] \* b[j];

/\*

\*这里是移位、进位

\*/

for (i = 0; i < 2 \* N - 1; ++i) {

c[i + 1] += c[i] / 10;//将十位上的数向前进位，并加上原来这个位上的数

c[i] = c[i] % 10;//将剩余的数存原来的位置上

}

}

int main() {

int a[N], b[N], c[2 \* N];

string s1, s2;

int j = 2 \* N - 1;

cout <<"input the first number:";

cin >> s1;

cout <<"input the second number:";

cin >> s2;

getdigits(a, s1);

getdigits(b, s2);

multiply(a, b, c);

while (c[j] == 0) {

if (j == 0) break;

j--;

}

for (int i = j; i >= 0; --i)

cout << c[i];

cout << endl;

return 0;

}

## 高精度加减乘除取余运算

说明：

1. 此算法可能超时。

1.比较

inlineint compare(stringstr1, stringstr2)

{

if (str1.size() >str2.size()) //长度长的整数大于长度小的整数

return 1;

elseif (str1.size() <str2.size())

return -1;

else

returnstr1.compare(str2); //若长度相等，从头到尾按位比较，compare函数：相等返回0，大于返回1，小于返回－1

}

2.高精度加法

string ADD\_INT(stringstr1, stringstr2)

{

string MINUS\_INT(string str1, string str2);

int sign = 1; //sign 为符号位

string str;

if (str1[0] == '-') {

if (str2[0] == '-') {

sign = -1;

str = ADD\_INT(str1.erase(0, 1), str2.erase(0, 1));

}

else {

str = MINUS\_INT(str2, str1.erase(0, 1));

}

}

else {

if (str2[0] == '-')

str = MINUS\_INT(str1, str2.erase(0, 1));

else {

//把两个整数对齐，短整数前面加0补齐

string::size\_type l1, l2;

int i;

l1 = str1.size(); l2 = str2.size();

if (l1 < l2) {

for (i = 1; i <= l2 - l1; i++)

str1 = "0" + str1;

}

else {

for (i = 1; i <= l1 - l2; i++)

str2 = "0" + str2;

}

int int1 = 0, int2 = 0; //int2 记录进位

for (i = str1.size() - 1; i >= 0; i--) {

int1 = (int(str1[i]) - 48 + int(str2[i]) - 48 + int2) % 10; //48 为 '0' 的ASCII 码

int2 = (int(str1[i]) - 48 + int(str2[i]) - 48 + int2) / 10;

str = char(int1 + 48) + str;

}

if (int2 != 0) str = char(int2 + 48) + str;

}

}

//运算后处理符号位

if ((sign == -1) && (str[0] != '0'))

str = "-" + str;

return str;

}

3.高精度减法

string MINUS\_INT(stringstr1, stringstr2)

{

string MULTIPLY\_INT(string str1, string str2);

int sign = 1; //sign 为符号位

string str;

if (str2[0] == '-')

str = ADD\_INT(str1, str2.erase(0, 1));

else {

int res = compare(str1, str2);

if (res == 0) return"0";

if (res < 0) {

sign = -1;

string temp = str1;

str1 = str2;

str2 = temp;

}

string::size\_type tempint;

tempint = str1.size() - str2.size();

for (int i = str2.size() - 1; i >= 0; i--) {

if (str1[i + tempint] <str2[i]) {

str1[i + tempint - 1] = char(int(str1[i + tempint - 1]) - 1);

str = char(str1[i + tempint] - str2[i] + 58) + str;

}

else

str = char(str1[i + tempint] - str2[i] + 48) + str;

}

for (int i = tempint - 1; i >= 0; i--)

str = str1[i] + str;

}

//去除结果中多余的前导0

str.erase(0, str.find\_first\_not\_of('0'));

if (str.empty()) str = "0";

if ((sign == -1) && (str[0] != '0'))

str = "-" + str;

return str;

}

4.高精度乘法

string MULTIPLY\_INT(stringstr1, stringstr2)

{

int sign = 1; //sign 为符号位

string str;

if (str1[0] == '-') {

sign \*= -1;

str1 = str1.erase(0, 1);

}

if (str2[0] == '-') {

sign \*= -1;

str2 = str2.erase(0, 1);

}

int i, j;

string::size\_type l1, l2;

l1 = str1.size(); l2 = str2.size();

for (i = l2 - 1; i >= 0; i--) { //实现手工乘法

string tempstr;

int int1 = 0, int2 = 0, int3 = int(str2[i]) - 48;

if (int3 != 0) {

for (j = 1; j <= (int)(l2 - 1 - i); j++)

tempstr = "0" + tempstr;

for (j = l1 - 1; j >= 0; j--) {

int1 = (int3 \* (int(str1[j]) - 48) + int2) % 10;

int2 = (int3 \* (int(str1[j]) - 48) + int2) / 10;

tempstr = char(int1 + 48) + tempstr;

}

if (int2 != 0) tempstr = char(int2 + 48) + tempstr;

}

str = ADD\_INT(str, tempstr);

}

//去除结果中的前导0

str.erase(0, str.find\_first\_not\_of('0'));

if (str.empty()) str = "0";

if ((sign == -1) && (str[0] != '0'))

str = "-" + str;

return str;

}

5.高精度除法

string DIVIDE\_INT(stringstr1, stringstr2, intflag)

{

//flag = 1时,返回商; flag = 0时,返回余数

string quotient, residue; //定义商和余数

int sign1 = 1, sign2 = 1;

if (str2 == "0") { //判断除数是否为0

quotient = "ERROR!";

residue = "ERROR!";

if (flag == 1) return quotient;

elsereturn residue;

}

if (str1 == "0") { //判断被除数是否为0

quotient = "0";

residue = "0";

}

if (str1[0] == '-') {

str1 = str1.erase(0, 1);

sign1 \*= -1;

sign2 = -1;

}

if (str2[0] == '-') {

str2 = str2.erase(0, 1);

sign1 \*= -1;

}

int res = compare(str1, str2);

if (res < 0) {

quotient = "0";

residue = str1;

}

elseif (res == 0) {

quotient = "1";

residue = "0";

}

else {

string::size\_type l1, l2;

l1 = str1.size(); l2 = str2.size();

string tempstr;

tempstr.append(str1, 0, l2 - 1);

//模拟手工除法

for (int i = l2 - 1; i < l1; i++) {

tempstr = tempstr + str1[i];

for (char ch = '9'; ch >= '0'; ch--) { //试商

string str;

str = str + ch;

if (compare(MULTIPLY\_INT(str2, str), tempstr) <= 0) {

quotient = quotient + ch;

tempstr = MINUS\_INT(tempstr, MULTIPLY\_INT(str2, str));

break;

}

}

}

residue = tempstr;

}

//去除结果中的前导0

quotient.erase(0, quotient.find\_first\_not\_of('0'));

if (quotient.empty()) quotient = "0";

if ((sign1 == -1) && (quotient[0] != '0'))

quotient = "-" + quotient;

if ((sign2 == -1) && (residue[0] != '0'))

residue = "-" + residue;

if (flag == 1) return quotient;

elsereturn residue;

}

//高精度除法,返回商

string DIV\_INT(stringstr1, stringstr2)

{

return DIVIDE\_INT(str1, str2, 1);

}

//高精度除法,返回余数

string MOD\_INT(stringstr1, stringstr2)

{

return DIVIDE\_INT(str1, str2, 0);

}

int main() {

char ch;

string s1, s2, res;

while (cin >> ch) {

cin >> s1 >> s2;

switch (ch) {

case'+': res = ADD\_INT(s1, s2); break; //高精度加法

case'-': res = MINUS\_INT(s1, s2); break; //高精度减法

case'\*': res = MULTIPLY\_INT(s1, s2); break; //高精度乘法

case'/': res = DIV\_INT(s1, s2); break; //高精度除法, 返回商

case'm': res = MOD\_INT(s1, s2); break; //高精度除法, 返回余数

default: break;

}

cout << res << endl;

}

return(0);

}

## 数组方式实现的高精度与整型加法

typedefint hp[250];

void Init(hp &a) {

string s;

int len;

memset(a, 0, sizeof(a));

cin >> s;

len = s.size();

for (int i = len; i >= 1; i--)

a[i] = s[len-i] - '0';

a[0] = len;

}

void print(hp a) {

for (int i = a[0]; i >= 1; i--)

cout << a[i];

cout << endl;

}

void add\_int(hp a, int x, hp &b) {

hp tmp;

memcpy(tmp, a, sizeof(tmp));

int len = 1;

tmp[len] += x;

while (tmp[len] > 9) {

tmp[len + 1] += tmp[len] / 10;

tmp[len] %= 10;

len++;

}

if (len > tmp[0]) tmp[0] = len;

memcpy(b, tmp, sizeof(b));

}

# 计算几何

## 几何公式

三角形:

1. 半周长 P=(a+b+c)/2

2. 面积 S=a\*Ha/2=absin(C)/2=sqrt(P(P-a)(P-b)(P-c))

3. 中线 Ma=sqrt(2(b^2+c^2)-a^2)/2=sqrt(b^2+c^2+2bccos(A))/2

4. 角平分线 Ta=sqrt(bc((b+c)^2-a^2))/(b+c)=2bccos(A/2)/(b+c)

5. 高线 Ha=bsin(C)=csin(B)=sqrt(b^2-((a^2+b^2-c^2)/(2a))^2)

6. 内切圆半径 r=S/P=asin(B/2)sin(C/2)/sin((B+C)/2)

=4Rsin(A/2)sin(B/2)sin(C/2)=sqrt((P-a)(P-b)(P-c)/P)

=Ptan(A/2)tan(B/2)tan(C/2)

7. 外接圆半径 R=abc/(4S)=a/(2sin(A))=b/(2sin(B))=c/(2sin(C))

四边形:

D1,D2为对角线,M对角线中点连线,A为对角线夹角

1. a^2+b^2+c^2+d^2=D1^2+D2^2+4M^2

2. S=D1D2sin(A)/2

(以下对圆的内接四边形)

3. ac+bd=D1D2

4. S=sqrt((P-a)(P-b)(P-c)(P-d)),P为半周长

正n边形:

R为外接圆半径,r为内切圆半径

1. 中心角 A=2PI/n

2. 内角 C=(n-2)PI/n

3. 边长 a=2sqrt(R^2-r^2)=2Rsin(A/2)=2rtan(A/2)

4. 面积 S=nar/2=nr^2tan(A/2)=nR^2sin(A)/2=na^2/(4tan(A/2))

圆:

1. 弧长 l=rA

2. 弦长 a=2sqrt(2hr-h^2)=2rsin(A/2)

3. 弓形高 h=r-sqrt(r^2-a^2/4)=r(1-cos(A/2))=atan(A/4)/2

4. 扇形面积 S1=rl/2=r^2A/2

5. 弓形面积 S2=(rl-a(r-h))/2=r^2(A-sin(A))/2

棱柱:

1. 体积 V=Ah,A为底面积,h为高

2. 侧面积 S=lp,l为棱长,p为直截面周长

3. 全面积 T=S+2A

棱锥:

1. 体积 V=Ah/3,A为底面积,h为高

(以下对正棱锥)

2. 侧面积 S=lp/2,l为斜高,p为底面周长

3. 全面积 T=S+A

棱台:

1. 体积 V=(A1+A2+sqrt(A1A2))h/3,A1.A2为上下底面积,h为高

(以下为正棱台)

2. 侧面积 S=(p1+p2)l/2,p1.p2为上下底面周长,l为斜高

3. 全面积 T=S+A1+A2

圆柱:

1. 侧面积 S=2PIrh

2. 全面积 T=2PIr(h+r)

3. 体积 V=PIr^2h

圆锥:

1. 母线 l=sqrt(h^2+r^2)

2. 侧面积 S=PIrl

3. 全面积 T=PIr(l+r)

4. 体积 V=PIr^2h/3

圆台:

1. 母线 l=sqrt(h^2+(r1-r2)^2)

2. 侧面积 S=PI(r1+r2)l

3. 全面积 T=PIr1(l+r1)+PIr2(l+r2)

4. 体积 V=PI(r1^2+r2^2+r1r2)h/3

球:

1. 全面积 T=4PIr^2

2. 体积 V=4PIr^3/3

球台:

1. 侧面积 S=2PIrh

2. 全面积 T=PI(2rh+r1^2+r2^2)

3. 体积 V=PIh(3(r1^2+r2^2)+h^2)/6

球扇形:

1. 全面积 T=PIr(2h+r0),h为球冠高,r0为球冠底面半径

2. 体积 V=2PIr^2h/3

## 判断一个点是否在三角形内

说明：

1. 不能用cmath里的abs，因为该abs返回的是整数；

2. 三角形的三个点的坐标存放在x[3]和y[3]中。

int x[3], y[3];

double abs(doublex) {

if (x>= 0) returnx;

elsereturn -x;

}

bool isIn(intxi, intyi) {

double S, S1, S2, S3;

S = abs((x[0]\*y[1]+x[1]\*y[2]+x[2]\*y[0]-y[0]\*x[1]-y[1]\*x[2]-y[2]\*x[0])/2.0);

S1 = abs((xi\*y[1]+x[1]\*y[2]+x[2]\*yi-yi\*x[1]-y[1]\*x[2]-y[2]\*xi)/2.0);

S2 = abs((x[0]\*yi+xi\*y[2]+x[2]\*y[0]-y[0]\*xi-yi\*x[2]-y[2]\*x[0])/2.0);

S3 = abs((x[0]\*y[1]+x[1]\*yi+xi\*y[0]-y[0]\*x[1]-y[1]\*xi-yi\*x[0])/2.0);

if (abs(S1+S2+S3-S) < 1e-7 && S1 != 0 && S2 != 0 && S3 != 0) returntrue;

elsereturnfalse;

## 判断点是否在线段上

说明：

Struct Lpoint {double x,y;}; //点

Struct Llineseg{Lpoint a,b;}; //线段

Struct Ldir{double dx,dy;}; //方向向量

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

| (P1-P0)\*(P2-P0)的叉积

若结果为正，则<P0,P1>在<P0,P2>的顺时针方向；

若为0则<P0,P1><P0,P2>共线；

若为负则<P0,P1>在<P0,P2>的在逆时针方向;

可以根据这个函数确定两条线段在交点处的转向,

比如确定p0p1和p1p2在p1处是左转还是右转，只要求

(p2-p0)\*(p1-p0)，若<0则左转，>0则右转，=0则共线

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double xmulti(Lpoint p1,Lpoint p2,Lpoint p0) {

return((p1.x-p0.x) \* (p2.y-p0.y) -

(p2.x-p0.x) \* (p1.y-p0.y));

}

int ponls(Llineseg l,Lpoint p) {

return( (xmulti(l.b,p,l.a)==0) &&// 该条件判断是否共线，如需判断点是否在直线上，只需删除下面两个判定即可

( ((p.x-l.a.x)\*(p.x-l.b.x)<0 ) ||

((p.y-l.a.y)\*(p.y-l.b.y)<0 )) ); // 后面两个条件是判断点p是否在线段两端点之间

}

## Graham求凸包（已修改）

说明：

1.该算法由《吉林大学ACM模板》（P34）提供;

2.该算法复杂度为O(N\*logN);

3.mult()叉乘函数，由<sp,ep>和<ep,op>叉乘

4.pnt[]是平面里的点集，n是点的个数，res[]是构成凸包的点集。

\*/

#include<algorithm>

struct Point { int x, y; };

bool mult(Point sp, Point ep, Point op) {

return (sp.x - op.x) \* (ep.y - op.y) >= (ep.x - op.x) \* (sp.y - op.y);

}

bool operator< (const Point &l, const Point &r) {

return l.y < r.y || (l.y == r.y && l.x < r.x);//对此排序，找到最下面，且同高度下最左边的点。

}

vector<Point> graham(vector<Point>pnt, int n) {

vector<Point>res;

int i, len, k = 0, top = 1;

sort(pnt.begin(), pnt.end());

if (n == 0) return res; res.push\_back(pnt[0]);

if (n == 1) return res; res.push\_back(pnt[1]);

if (n == 2) return res; res.push\_back(pnt[2]);

// 不断加入新的点，并通过叉积判断是否为凸包上的点

for (i = 2; i < n; i++) { // pnt[i]是当前被遍历的点

while (top && mult(pnt[i], res[top], res[top - 1])) // res[top]和res[top-1]是前两个点

top--; // 若判断得前面一个点不符合，前一个点移除（这里可能会移除掉后面要用的点）

++top;

res.resize(top+1); // 无论前一个点是否被移除，新点都补上

res[top] = pnt[i];

}

// 接下来的操作是为了补全凸包（前面的操作可能会把一些之后要用的点给pop掉了）

len = top;

top++;

res.resize(top+1);

res[top] = pnt[n - 2]; // 此时len已经是新栈的栈底

for (i = n - 3; i >= 0; i--) {

while (top != len && mult(pnt[i], res[top], res[top - 1])) // top!=len相当于前面的top!=0

top--;

++top;

res.resize(top+1);

res[top] = pnt[i];

}

return res; // 返回凸包数组

}

## }点到线段的最短距离

double PointToSegDist(double x, double y, double x1, double y1, double x2, double y2)

{

double cross = (x2 - x1) \* (x - x1) + (y2 - y1) \* (y - y1);

if (cross <= 0) return sqrt((x - x1) \* (x - x1) + (y - y1) \* (y - y1));

double d2 = (x2 - x1) \* (x2 - x1) + (y2 - y1) \* (y2 - y1);

if (cross >= d2) return sqrt((x - x2) \* (x - x2) + (y - y2) \* (y - y2));

double r = cross / d2;

double px = x1 + (x2 - x1) \* r;

double py = y1 + (y2 - y1) \* r;

return sqrt((x - px) \* (x - px) + (py - y1) \* (py - y1));

}

## 空间两线段的最短距离

float DistanceLineToLine( const osg::Vec3d& p1,const osg::Vec3d& p2,const osg::Vec3d& p3,const osg::Vec3d& p4 )

{

float distance;

float x1 = p1.x(); //A点坐标（x1,y1,z1）

float y1 = p1.y();

float z1 = p1.z();

float x2 = p2.x(); //B点坐标（x2,y2,z2）

float y2 = p2.y();

float z2 = p2.z();

float x3 = p3.x(); //C点坐标（x3,y3,z3）

float y3 = p3.y();

float z3 = p3.z();

float x4 = p4.x(); //D点坐标（x4,y4,z4）

float y4 = p4.y();

float z4 = p4.z();

float a = (x2-x1)\*(x2-x1)+(y2-y1)\*(y2-y1)+(z2-z1)\*(z2-z1);

float b = -((x2-x1)\*(x4-x3)+(y2-y1)\*(y4-y3)+(z2-z1)\*(z4-z3));

float c = -((x1-x2)\*(x1-x3)+(y1-y2)\*(y1-y3)+(z1-z2)\*(z1-z3));

float d = -((x2-x1)\*(x4-x3)+(y2-y1)\*(y4-y3)+(z2-z1)\*(z4-z3));

float e = (x4-x3)\*(x4-x3)+(y4-y3)\*(y4-y3)+(z4-z3)\*(z4-z3);

float f = -((x1-x3)\*(x4-x3)+(y1-y3)\*(y4-y3)+(z1-z3)\*(z4-z3));

if ((a\*e-b\*d)==0&&(b\*d-a\*e)==0) //平行

{

float d1 = (p1-p3).length();

float d2 = (p1-p4).length();

distance = (d1<d2)?d1:d2;

return distance;

}

float s = (b\*f-e\*c)/(a\*e-b\*d);

float t = (a\*f-d\*c)/(b\*d-a\*e);

if(0<=s&&s<=1&&0<=t&&t<=1) //说明P点落在线段AB上,Q点落在线段CD上

{

//2条线段的公垂线段PQ;

//P点坐标

float X = x1+s\*(x2-x1);

float Y = y1+s\*(y2-y1);

float Z = z1+s\*(z2-z1);

//Q点坐标

float U = x3+t\*(x4-x3);

float V = y3+t\*(y4-y3);

float W = z3+t\*(z4-z3);

osg::Vec3d P(X,Y,Z);

osg::Vec3d Q(U,V,W);

distance = (P-Q).length();

}

else

{

float d1 = DistancePointToLine(p3,p4,p1);

float d2 = DistancePointToLine(p3,p4,p2);

float d3 = DistancePointToLine(p1,p2,p3);

float d4 = DistancePointToLine(p1,p2,p4);

distance = (d1<d2)?d1:d2;

distance = (distance<d3)?distance:d3;

distance = (distance<d4)?distance:d4;

}

return distance;

}

## 空间点到线段的最短距离

float DistancePointToLine( const osg::Vec3d& star, const osg::Vec3d& end,const osg::Vec3d& center )

{

float distance;

float x0 = center.x();//P点坐标（x0,y0,z0）

float y0 = center.y();

float z0 = center.z();

float x1 = star.x(); //A点坐标（x1,y1,z1）

float y1 = star.y();

float z1 = star.z();

float x2 = end.x();//B点坐标（x2,y2,z2）

float y2 = end.y();

float z2 = end.z();

float t = ((x1-x0)\*(x1-x2)+(y1-y0)\*(y1-y2)+(z1-z0)\*(z1-z2))

/((x1-x2)\*(x1-x2)+(y1-y2)\*(y1-y2)+(z1-z2)\*(z1-z2));

if (0<=t&&t<=1)//垂足Q点落在线段AB上

{

float X = x1+t\*(x2-x1);

float Y = y1+t\*(y2-y1);

float Z = z1+t\*(z2-z1);

osg::Vec3d Q(X,Y,Z);

distance = (Q-center).length();

}

if (t<0) //垂足Q点不落在线段AB上，而是落在BA的延长线上

{

distance = (star-center).length();

}

if (t>1) //垂足Q点不落在线段AB上，而是落在AB的延长线上

{

distance = (end-center).length();

}

return distance;

}

# 数据结构

## 二分查找

int binSearch(int \*arr,intleft, intright, intkey){

int mid;

while(left<= right){

mid = (left+right) / 2;

if(key == arr[mid]) return mid;

elseif(key<arr[mid]) right = mid-1;

elseif(key>arr[mid]) left = mid+1;

}

return -1;

}

## 并查集

#define MAXN 100005

int fa[MAXN] = {0};

int ranks[MAXN] = {0};

void initialise(int n) //初始化

{

for (int i = 1; i <= n; i++)

fa[i] = i,ranks[i] = 1;

}

int getfather(int v) //父节点

{

return (fa[v] == v) ? v : fa[v] = getfather(fa[v]);

}

void merge(int x,int y) //合并

{

x = getfather(x);

y = getfather(y);

if (x != y)

fa[x] = y,ranks[y] += ranks[x];

}

}

## 树状数组

#define LL long long

LL a[maxn];

LL lowbit(LLp) { return (p&-p); }

LL sum(LLp) {

LL ret = 0;

while (p> 0) ret+=a[p], p-=lowbit(p);

return ret;

}

void add(LLp, LLv) { // 若要减去，则v传入一个负数

while (p<= n) a[p]+=v, p+=lowbit(p);

}

## 线段树(部分修改)

### 线段树的结构体和命令行（已修改）

#define LL long long

struct node

{

LL lef;

LL rig;

LL mid;

LL data;

LL lazyn;

}tree[4 \* MAX];

vector<LL>sto;

### 线段树的建树（已修改）

LL createtree(LL left, LL right, LL id)//建树

{

tree[id].lef = left;//划定区间

tree[id].rig = right;

tree[id].lazyn = 0;//初始化lazy标记

if (left == right)//叶节点创建

{

tree[id].data = sto[left];

return tree[id].data;

}

LL mid = (left + right) >> 1;//找到中值

tree[id].mid = mid;//避免重复计算

tree[id].data = createtree(left, mid, id<<1) + createtree(mid + 1, right, id<<1|1);//区间求和情况，根据不同可能需要改动

return tree[id].data;

}

### 线段树区间查询（已修改）

（以区间求和为例，求最值需做修改）

LL query(LL lef, LL rig, LL id)

{

if (tree[id].lef >= lef&&tree[id].rig <= rig)//类似于update，但是有些题目会在是否要在查找时pushdown这个问题上卡时间

{

//cout << tree[id].lef << " " << tree[id].rig << " " << tree[id].data << "|";

return tree[id].data;

}

pushdown(id);

if (lef > tree[id].mid)

return query(lef, rig,id<<1|1);

else if (tree[id].mid >= rig)

return query(lef, rig,id<<1);

else

return query(lef, rig,id<<1|1) + query(lef, rig,id<<1);

}

### 线段树的区间更新（已修改）

1. 值增加（风格1）

void update(LL left, LL right, LL value, LL id)//因为一开始的left和right必然在目录中，所以可以用不断二分的方法求出其区间和

{

if (tree[id].lef >= left&&tree[id].rig <= right{

tree[id].data += (tree[id].rig - tree[id].lef + 1)\*value;//如果在所求区间内，则更新点存数据

tree[id].lazyn += value;//更新暂存数据

//cout << tree[id].lef << " " << tree[id].rig << " " << tree[id].data << "|";

return;

}

pushdown(id);//如果不在所求区间内，那么为了数据的准确，该节点暂存数据下放

if (left > tree[id].mid)//为了不出现越界的情况，数据必须准确，如果一个节点不属于所求范围内，就会不断的放大和缩小，最终爆数组造成RTE

{

update(left, right, value,id<<1|1);

}

else if (tree[id].mid >= right)

{

update(left, right, value,id<<1);

}

else

{

update(left, right, value,id<<1);

update(left, right, value,id<<1|1);

}

tree[id].data = tree[id<<1].data + tree[id<<1|1].data;

}

2.值覆盖（风格2）

inline void update(int rt,int left,int right,int value) //区间更新

{

if(left==node[rt].l && node[rt].r==right)

{

node[rt].val = value;

node[rt].sum = value\*(node[rt].r-node[rt].l+1);

return;

}

if(node[rt].val) //判断当前节点是否有做过lazy处理

{

node[rt\*2].sum = node[rt].val\*(node[rt\*2].r-node[rt\*2].l+1);

node[rt\*2+1].sum = node[rt].val\*(node[rt\*2+1].r-node[rt\*2+1].l+1);

node[rt\*2].val = node[rt].val;

node[rt\*2+1].val = node[rt].val;

node[rt].val = 0;

}

int mid = (node[rt].l+node[rt].r)/2;

if(left > mid) // 更新右子树

update(rt\*2+1,left,right,value);

else if(right <= mid) //更新左子树

update(rt\*2,left,right,value);

else // 更新左右子树

{

update(rt\*2,left,mid,value);

update(rt\*2+1,mid+1,right,value);

}

node[rt].sum = node[rt\*2].sum+node[rt\*2+1].sum;

}

### 线段树的区间合并

题目大意：

输入 1 a:询问是不是有连续长度为a的空房间,有的话住进最左边，并输出最左端编号

输入 2 a b:将[a,a+b-1]的房间清空

思路：因为是统计存在的最长连续房间，左右将值分成了三段，计算的时候分三个，即纯左孩子剩余间输中计算和纯右孩子剩余间输中计算，还有一左一右。就三种

#include <iostream>

#include <cstdio>

#include <algorithm>

using namespace std;

const int N = 50005;

/\*线段树操作:update:区间替换 query:询问满足条件的最左端点\*/

struct Node

{

int l,r;

int lsum,rsum,msum;

int ck;

int mid(){return (l+r)>>1;}

}tree[N<<2];

void PushDown(int rt)

{

if(tree[rt].ck != -1)

{

int ll = tree[rt<<1].r - tree[rt<<1].l + 1;

int rr = tree[rt<<1|1].r - tree[rt<<1|1].l + 1;

tree[rt<<1].ck = tree[rt<<1|1].ck = tree[rt].ck;

tree[rt].ck = -1;

tree[rt<<1].rsum = tree[rt<<1].lsum = tree[rt<<1].msum = tree[rt<<1].ck ?0:ll;

tree[rt<<1|1].rsum = tree[rt<<1|1].lsum = tree[rt<<1|1].msum = tree[rt<<1|1].ck ?0:rr;

}

}

void PushUp(int rt)

{

int ll = tree[rt<<1].r - tree[rt<<1].l + 1;

int rr = tree[rt<<1|1].r - tree[rt<<1|1].l + 1;

tree[rt].lsum = tree[rt<<1].lsum;

if(tree[rt<<1].lsum == ll) tree[rt].lsum += tree[rt<<1|1].lsum;

tree[rt].rsum = tree[rt<<1|1].rsum;

if(tree[rt].rsum == rr) tree[rt].rsum += tree[rt<<1].rsum;

tree[rt].msum= max(max(tree[rt<<1].msum,tree[rt<<1|1].msum),tree[rt<<1].rsum+tree[rt<<1|1].lsum);

}

void build(int l,int r,int rt)

{

tree[rt].l = l,tree[rt].r = r,tree[rt].ck = -1;

tree[rt].lsum = tree[rt].rsum = tree[rt].msum = r-l+1;

if(l == r) return;

int m = (l+r)>>1;

build(l,m,rt<<1);

build(m+1,r,rt<<1|1);

PushUp(rt);

}

void update(int l,int r,int c,int rt)

{

if(tree[rt].l == l && tree[rt].r == r)

{

tree[rt].msum = tree[rt].lsum = tree[rt].rsum = c?0:(r-l+1);

tree[rt].ck = c;

return;

}

PushDown(rt);

int m = tree[rt].mid();

if(r<=m) update(l,r,c,rt<<1);

else if(l>m) update(l,r,c,rt<<1|1);

else

{

update(l,m,c,rt<<1);

update(m+1,r,c,rt<<1|1);

}

PushUp(rt);

}

int query(int w,int rt)

{

if (tree[rt].l == tree[rt].r) return tree[rt].l;

PushDown(rt);

int m = tree[rt].mid();

if (tree[rt<<1].msum >= w)//如果左子树的最大连续空>=需求量，那么直接进入左子树，=也去左子树的原因是题目要求的最左

return query(w , rt<<1);

//中间的

else if (tree[rt<<1].rsum + tree[rt<<1|1].lsum >= w)//左子树的连续右+右子树的连续左>=w，说明找到了可以直接求出

return m - tree[rt<<1].rsum + 1;

return query(w , rt<<1|1);

}

int main()

{

int n,m;

while(~scanf("%d %d",&n,&m))

{

build(1,n,1);

while(m--)

{

int op,a,b;

scanf("%d",&op);

if(op == 1)

{

scanf("%d",&a);

if(tree[1].msum < a) puts("0");//根节点的最大连续空间不够

else

{

int p = query(a,1);

printf("%d\n",p);

update(p,p+a-1,1,1);//把这段更新为被覆盖

}

}

else

{

scanf("%d %d",&a,&b);

update(a,a+b-1,0,1);//把这段更新为未被覆盖

}

}

}

return 0;

}

### 二维线段树

代码：(求区间最大为例，每次在一个点加上某个值，初试状态矩阵为0)

usingnamespacestd;

constint INF = 0;

constint MAXN = 501;

struct Nodey

{

int l,r;

longlong Max;

};

int locy[MAXN],locx[MAXN];,e

struct Nodex

{

int l,r;

Nodey sty[MAXN\*4];

void build(int i,int \_l,int \_r) {

sty[i].l = \_l;

sty[i].r = \_r;

sty[i].Max = INF;

if(\_l == \_r){locy[\_l] = i,return; }

int mid = (\_l + \_r)/2;

build(i<<1,\_l,mid);

build((i<<1)+1,mid+1,\_r);

}

longlong queryMax(longlong i,int \_l,int \_r)

{

if(sty[i].l == \_l &&sty[i].r == \_r)

returnsty[i].Max;

int mid = (sty[i].l + sty[i].r)/2;

if(\_r <= mid)

returnqueryMax(i<<1,\_l,\_r);

elseif(\_l > mid)

returnqueryMax((i<<1)+1,\_l,\_r);

else

return max(queryMax(i<<1,\_l,mid),queryMax((i<<1)+1,mid+1,\_r));}

}stx[MAXN\*4];

int n;

void build(int i,int l,int r){

stx[i].l = l;

stx[i].r = r;

stx[i].build(1,1,n);

if(l == r){

locx[l] = i;

return;}

int mid = (l+r)/2;

build(i<<1,l,mid);

build((i<<1)+1,mid+1,r);

}

//修改值

void Modify(int x,int y,longlong val){

int tx = locx[x];

int ty = locy[y];

stx[tx].sty[ty].Max += val;

for(int i = tx;i >0;i >>= 1)

for(int j = ty;j >0;j >>= 1){

if(i == tx && j == ty)

continue;

if(j == ty)

stx[i].sty[j].Max = max(stx[i<<1].sty[j].Max,stx[(i<<1)+1].sty[j].Max);

else

stx[i].sty[j].Max = max(stx[i].sty[j<<1].Max,stx[i].sty[(j<<1)+1].Max);}}

longlong queryMax(longlong i,int x1,int x2,int y1,int y2){

if(stx[i].l == x1 &&stx[i].r == x2)

returnstx[i].queryMax(1,y1,y2);

int mid = (stx[i].l + stx[i].r)/2;

if(x2 <= mid)

return queryMax(i<<1,x1,x2,y1,y2);

elseif(x1 > mid)

return queryMax((i<<1)+1,x1,x2,y1,y2);

else

return max(queryMax(i<<1,x1,mid,y1,y2),queryMax((i<<1)+1,mid+1,x2,y1,y2));}

int main()

{

int Q,M;

while(cin>>n>> Q >> M) {

build(1,1,n);

while(M--) {

int x1,y1,x2,y2;

scanf("%d %d %d %d",&x1,&y1,&x2,&y2);

int value;

for(int i = x1;i <= x2;i++) {

for(int j = y1;j <= y2;j++) {

scanf("%d" , &value);

Modify(i, j, value); } } }

while(Q--){

int I,J,S;

scanf("%d %d %d",&I,&J,&S);

printf("%lld\n",queryMax(1, I, I+S-1, J, J+S-1));}}

return0;}

## 拓扑排序

/\* 拓扑排序(邻接阵形式) \*/

/\*

说明：

1. 拓扑排序,邻接阵形式,复杂度O(n^2)

2. 如果无法完成排序,返回0,否则返回1,ret返回有序点列

3. 传入图的大小n和邻接阵mat,不相邻点边权0

\*/

#define MAXN 100

int toposort(intn,intmat[][MAXN],int\* ret){

int d[MAXN],i,j,k;

for (i=0;i<n;i++)

for (d[i]=j=0;j<n;d[i]+=mat[j++][i]);

for (k=0;k<n;ret[k++]=i){

for (i=0;d[i]&&i<n;i++);

if (i==n)

return 0;

for (d[i]=-1,j=0;j<n;j++)

d[j]-=mat[i][j];

}

return 1;

}

## 单调队列

#include <deque>

int a[MAX]=1000;

deque<int>monoQueue;//monotonic queue

void push(int x)//传递的是下标

{

while(!monoQueue.empty() && a[monoQueue.back()]>a[x])//这里是递增队列，改成递减大于变小于即可

monoQueue.pop\_back();

monoQueue.push\_back(x);

}

每次移动窗时

if(monoQueue.front()<窗长度)monoQueue.pop\_front();

push(i);//然后把新元素加进去

## AVL Tree (链表形式)

template <typename type>

class AVL

{

private:

class node

{

public:

type data;

int bf, h, size;

node \*l, \*r;

node() :l(NULL), r(NULL), bf(0), h(1), size(1) {};

node(type data) :l(NULL), r(NULL), bf(0), h(1), data(data), size(1) {};

int geth()

{

if (!this)return 0;

else return h;

}

int getsize()

{

if (!this)return 0;

else return size;

}

void update()

{

if (!this)return;

int lh, rh;

lh = l->geth();

rh = r->geth();

bf = lh - rh;

h = max(lh, rh) + 1;

size = l->getsize() + r->getsize() + 1;

}

}\*root;

void Lrotation(node \* &p)

{

node \*temp = p->l;

p->l = temp->r;

temp->r = p;

p = temp;

p->r->update();

p->update();

}

void Rrotation(node \* &p)

{

node \*temp = p->r;

p->r = temp->l;

temp->l = p;

p = temp;

p->l->update();

p->update();

}

void balance(node \* &p)

{

if (!p)return;

if (p->bf >= 2)

{

int lh = p->l->l->geth();

int rh = p->l->r->geth();

if (lh >= rh)

Lrotation(p);

else

{

Rrotation(p->l);

Lrotation(p);

}

}

if (p->bf <= -2)

{

int lh = p->r->l->geth();

int rh = p->r->r->geth();

if (rh >= lh)

Rrotation(p);

else

{

Lrotation(p->r);

Rrotation(p);

}

}

}

void insert(node \* &p, type data)

{

if (!p) {

p = new node(data);

return;

}

if (data <= p->data)

insert(p->l, data);

else

insert(p->r, data);

p->update();

balance(p);

}

void erase(node \* &p, type data)

{

if (!p)return;

if (p->data == data)

{

if (p->l && p->r)

{

node \*temp = p->r;

while (temp->l)

temp = temp->l;

p->data = temp->data;

if (p->r->l)

{

temp = p->r;

while (temp->l->l)

temp = temp->l;

delete temp->l;

temp->l = NULL;

}

else

erase(p->r, p->data);

stack<node \*>st;

st.push(p->r);

while (st.top()->l)

st.push(st.top()->l);

while (!st.empty())

{

st.top()->update();

st.pop();

}

}

else if (p->l)

{

node \*temp = p->l;

delete p;

p = temp;

}

else if (p->r)

{

node \*temp = p->r;

delete p;

p = temp;

}

else

{

delete p;

p = NULL;

}

}

else if (p->data < data)

erase(p->r, data);

else

erase(p->l, data);

p->update();

balance(p);

}

void clear(node \*p)

{

if (!p)return;

clear(p->l);

clear(p->r);

delete p;

}

void print(node \*p, type last)

{

if (!p)return;

cout << p->data << ":(" << last << ") ";

print(p->l, p->data);

print(p->r, p->data);

}

public:

AVL() :root(NULL) {};

void insert(type data)

{

insert(root, data);

}

void erase(type data)

{

erase(root, data);

}

int size()

{

return root->getsize();

}

void clear()

{

clear(root);

root = NULL;

}

void print()

{

if (root)

print(root, root->data);

else

cout << "no data" << endl;

}

};

# 图论

## Dijkstra

说明：

1. 该算法复杂度为n^2，可被优化成mlogn（m在稠密图时会接近n^2）

1. 读入权重时需注意如果是无向图则正反两向都需要赋值。

#include<cstring>

#define V (100+10) // 结点个数

#define E (10000+10) // 边的个数

#define INF V\*E+10

int d[V], w[V][V]; // 分别为最短路和某条边的权重

bool v[V]; // 这个结点有没访问过

void init() { // 将三个数组初始化

memset(v, 0, sizeof(v));

for (int i=0; i<n; i++) d[i] = INF;

for (int i=0; i<n; i++) for (int j=0; j<n; j++) w[i][j] = INF;

}

void dijkstra() {

for (int i=0; i<n; i++) {

int x, m = INF; // x用于保存与0距离最短的结点，m用于保存0到x的路径

for (int y=0; y<n; y++) // 获取与0距离最短的结点

if (!v[y] && d[y]<=m) m = d[x=y];

if(d[x]==INF)break;//找不到连通点

v[x] = 1; // 开始对这个点进行遍历

for (int y=0; y<n; y++) // 求出当前0到与x相邻点所有点的最短路径

d[y] = min(d[y], d[x]+w[x][y]);

}

}//初始化时将d[begin]=0

## Floyd

说明：

1.在稠密图时比Dijkstra有优势；

2.k在最外层，是为了以k为中介点，里面两个for循环则是进行全图更新；

3.每次至少能将k周边的点之间的距离更新，当选到最后一个k点时，进行全图更新时可借助之前求出的结果将全图的最短路更新出来；

4.该算法在改成最长路时，若图带环，则d[i][i]为环绕一次的最长路；

5.根据实际情况1可能要换成0。

for (int k=1; k<=n; k++)

for (int i=1; i<=n; i++)

for (int j=1; j<=n; j++)

if (g[i][k]<INF && g[k][j]<INF)

g[i][j] = min(g[i][j], g[i][k]+g[k][j]);

## 传递闭包判断图的连通性

说明：

1.该算法由Floyd算法改编而来。

for (int k=1; k<=n; k++)//注意1和n

for (int i=1; i<=n; i++)

for (int j=1; j<=n; j++)

d[i][j] |= d[i][j]||(d[i][k]&&d[k][j]);

## 树的最小支配集

贪心做法，最小支配集是用点来覆盖所有的点

描述：深搜，如果自身和父亲和孩子节点都没有被覆盖，那么将父亲覆盖

题目要求是安放的点输入1，不安放的输入0，是无向图

bool vis[N],set[N];//要清零，set[i]就是点i在不在支配集内，根节点from设为-1

void dfs(int u, int from)

{

vis[u]=1;

bool flag=0;

for(int i=head[u] ; ~i ; i=g[i].next)

{

int v=g[i].v;

if(!vis[v])

{

dfs(v,u);

flag|=set[v];

}

}

if(from==-1)

{

if(!flag)set[u]=1;

}

else if(!flag && !set[u] && !set[from])

{

set[from]=1;

}

}

## Hash

思路：通过将一个字符串通过一个转换函数，转换成一个下标来一一对应

具体做法：

定义一个结构体数组，数组下标为哈希值

用vector<struct\_name> name[MAX] 来防止哈希值冲突

举例说明：

#define mod 100003

#define ll long long

struct SHOP

{

int money;

string name;

};

vector<SHOP> shop[100010];

inline int myhash(string name) //获得hash下标

{

ll ans = 0;

for(int i = 0;i < name.size();i++)

ans = (ans\*1003 + (name[i]-'0')) % mod;

return (int)ans;

}

## 匈牙利二分图最大匹配

#include <iostream>

#include <cmath>

#include <cstring>

using namespace std;

const int MAX=110;

int m,n,a[MAX],b[MAX];

int vis[MAX],fa[MAX];//vis[i]表示此次寻找曾试图将i腾出来

bool path(int i)

{

for(int j=1 ; j<=m ; ++j)

{

if(两顶点连通 && vis[j]==0)//尝试每条边

{

vis[j]=1;

if(fa[j]==0 || path(fa[j]) )//如果还没被匹配 或者 可以腾出来

{

fa[j]=i;

return 1;//能腾出来

}

}

}

return 0;//找遍所有仍腾不出来

}

int main(void)

{

cin>>n;

for(int i=1 ; i<=n ; ++i)cin>>a[i];

cin>>m;

for(int i=1 ; i<=m ; ++i)cin>>b[i];

int count=0;

for(int i=1 ; i<=n ; ++i)

{

memset(vis,0,sizeof vis);//每次记得清零

if(path(i))count++;

}

cout<<count<<"\n";

return 0;

}

## 并查集

#include <iostream>

#include <cstdio>

#include <cstring>

using namespace std;

const int N=1000+10;

int pre[N];

int find(int x)

{

int r=x;

while(pre[r]!=r)

r=pre[r];

while(pre[x]!=x)

{

int t=pre[x];

pre[x]=r;

x=t;

}

return r;

}

void mix(int a, int b)

{

int fa=find(a),fb=find(b);

if(fa!=fb)

pre[fa]=fb;

}

int isHead[N];

int main()

{

int n,m;

while(scanf("%d",&n),n)

{

scanf("%d",&m);

memset(isHead,0,sizeof isHead);

for(int i=1 ; i<=n ; ++i)

{

pre[i]=i;

}

for(int i=0 ; i<m ; ++i)

{

int a,b;

scanf("%d%d",&a,&b);

mix(a,b);

}

for(int i=1 ; i<=n ; ++i)

{

isHead[find(i)]=1;

}

int cnt=0;

for(int i=1 ; i<=n ; ++i)

{

if(isHead[i])

cnt++;

}

printf("%d\n",cnt-1);

}

return 0;

}

## 最小生成树

### prim算法

//历遍vn中的所有点，找出与vn中点相连的最短边，并把这条边的另一端点加入vn，直到vn=v

//时间复杂度V^2，据说用邻接表可以弄成ElogV

//优化后的prim可读性太差，建议用kruskal

const int N=500+10;

const int INF=INT\_MAX;//注意相加溢出

int g[N][N];

bool vis[N];//标记是否已经选中

int dis[N];//记录每个点到vn中点的长度最小值

int to[N];//记录dis对应的边指向的vn中的点

int ans=0;//ans存的是边长总值

void prim(int n)

{

memset(vis,0,sizeof vis);

int p=1;

int now=0;//起点

vis[now]=1;

for(int i=0 ; i<n ; ++i)

{

dis[i]=g[now][i];

to[i]=now;

}

while(p!=n)

{

int m=INF;

for(int j=0 ; j<n ; ++j)//找出连接选中点和未选中点的最短边

{

if(!vis[j] && dis[j]<m)

{

m=dis[j];

now=j;

}

}

//如果要记录边的话记录<to[now],now>即可

p++;

vis[now]=1;//选中这个点

ans+=m;//累积边长

for(int i=0 ; i<n ; ++i)//更新dis,每次判断新加进来的点即可

{

if(!vis[i] && g[now][i]<dis[i])

{

dis[i]=g[now][i];

to[i]=now;

}

}

}

}//读图时务必将不联通的边权置为INF!

### kruskal算法

#include <bits/stdc++.h>

using namespace std;

//从小到大历遍所有边，如果边两端点未连通，就选中这条边，直到所有点连通

//时间复杂度ElogE+E，貌似比prim快

const int N=500;

const int E=N\*(N-1)/2;

struct edge

{

int u;

int v;

int w;

}

eg[E];//边集

bool cmp\_by\_w(edge a, edge b)

{

return a.w<b.w;

}

int vest[N];//集合编号，判断是否连通

int len\_tree=0;//len\_tree记录总边长

void kruskal(int n,int e)//使用前一定要排好序

{

for(int i=0 ; i<n ; ++i)vest[i]=i;//初始化vest

for(int i=0 ; i<e ; ++i)//从小到大遍历

{

edge &now=eg[i];

int sn1=vest[now.u];

int sn2=vest[now.v];

if(sn1!=sn2)//如果两顶点未连通

{

//将<now.u,now,v>(eg[i])选中

len\_tree+=now.w;

for(int i=0 ; i<n ; ++i)

{

if(vest[i]==sn2)//归并集合

vest[i]=sn1;

}

}

}

}

int main()

{

int T;

scanf("%d",&T);

while(T--)

{

len\_tree=0;

int n,e;

scanf("%d%d",&n,&e);

for(int i=0 ; i<e ; ++i)

{

int a,b,k;

scanf("%d%d%d",&a,&b,&k);

eg[i].u=a;

eg[i].v=b;

eg[i].w=k;

}

sort(eg,eg+e,cmp\_by\_w);//使用前一定要排好序

kruskal(n,e);

printf("%d\n",len\_tree);

}

return 0;

}

## tarjan算法找割点/割边

vector<int> G[maxn];//保存一个点的邻接点

int ans, low[maxn], pre[maxn], sum, root;//low为能接触到的最早的节点，pre为每个节点的访问顺序

bool iscut[maxn];//存储该点是否为割点

void dfs(int u, int f)//一个为现在点，一为现在点的父节点

{

int lowu;

low[u] = pre[u] = sum++;

int l = G[u].size();

int child = 0;

for (int i = 0; i<l; i++)

{

int v = G[u][i];

if (!pre[v]) {

child++;

dfs(v, u);

low[u] = min(low[u], low[v]);

if (u == root&&child == 2) iscut[u] = 1;

if (low[v] >= pre[u] && u != root) {

iscut[u] = 1;

}

}

else if (v != f)

{

low[u] = min(low[u], pre[v]);

}

}

}

//（未测试）求割边时类似，就是条件设置为low[v]>pre[u],即v子节点无法通过任何除了u->v以外的路径访问到更早的节点，那么这个路径一旦断掉必然产生割裂，即为割边。

# 动态规划

## 0-1背包

int dp[maxn + 1];

void solve()

{

for (int i = 0; i < n; i++)

{

for (int j = w; j>= w[i]; j--)

{

dp[j] = max(dp[j], dp[j - w[i]] + v[i]);

}

cout << dp[w] <<endl;

}

## 一个序列最少能被划分成多少个递增子序列

for (int i = 0; i < n; i++) {

dp[i] = 1;

for (int j = i; j >= 0; j--)

if (woods[i].w < woods[j].w) // 此处若改成>,则变成了求最长递增子序列的代码

dp[i] = max(dp[i], dp[j] + 1);}// dp数组中值相同的表示在同一个序列中，dp中的值最大到多少即为能被分成的最少组数

## 最长递增子序列

int n;

int dp[maxn], a[maxn]; // dp数组储存的是以该数字结尾的最长递增子序列

void solve(){

int res = 0;

for (int i = 0; i < n; i++) {

dp[i] = 1;

for (int j = 0; j < i; j++) {

if (a[i] > a[j]) dp[i] = max(dp[j]+1, dp[i]);

}

res = max(res,dp[i]);

}

cout << ans << endl;

}

## 最长公共子序列（modified O(nlogn)）

int a[100070], dp[100070], n;//n为数组长度，dp和a都要从1开始计数，原因是为了方便使得ans成为答案。

int LIS(int \*a)

{

int i, j, ans = 1;

dp[1] = a[1];

for (i = 2; i <= n; i++)

{

if (a[i] <= dp[1])//如果比最小的还小

j = 1;

else if (a[i] > dp[ans])//如果比最大的还大

j = ++ans;

else

j = lower\_bound(dp + 1, dp + ans, a[i]) - dp;

dp[j] = a[i];

}

return ans;

}

## 最长回文子序列

#define MAXN 1005

int a[MAXN<<1],dp[MAXN<<1][MAXN<<1];

int main() {

int n,i,j,l,ans;

while (~scanf("%d",&n) && n) {

for (i=1; i<=n; i++) scanf("%d",&a[i]),a[i+n]=a[i];

memset(dp,0,sizeof(dp));

for (i=1; i<=(n<<1); i++) dp[i][i]=1;

for (l=2; l<=n; l++) for (i=1; i<=n; i++) {

j=i+l-1;

if (j>(n<<1)) continue;

if (a[i]==a[j]) dp[i][j]=dp[i+1][j-1]+2;

else dp[i][j]=dp[i+1][j-1];

dp[i][j]=max(dp[i][j],max(dp[i+1][j],dp[i][j-1]));

}

ans=0;

for (i=1;i<=n;i++)

ans=max(ans,max(dp[i][i+n-1],dp[i][i+n-2]+1));

printf("%d\n",ans);

}

return 0;

}

# 自家模板

## KMP模板

void reset(string shift)

{

next\_store.clear();

start.clear();

next\_store.resize(shift.size(), 0);

next\_store[0] = 0;

int num = shift.size();

for (int i = 0; i < num - 1; i++)

{

if (shift[i + 1] == shift[next\_store[i]])

next\_store[i + 1] = next\_store[i] + 1;

}

}

void kmp(string shift,string text)

{

reset(shift);

int num\_t = text.size();

int num\_s = shift.size();

int flag = 0;

int q = -1;

for (int i = 0; i < num\_t; i++)

{

while (q > -1 && shift[q + 1] != text[i])//我们事实上不是在匹配q的所在字符，而是在匹配其下一位，所以q要+1

q = next\_store[q] - 1;//若匹配不成功，则返回到最优位置

if (shift[q + 1] == text[i])//如果没有到底部且匹配成功则转到下一个位置

q++;

if (q == num\_s - 1)//字符串匹配成功

{

if (flag)

cout << " ";

else

flag = 1;

printf("%d", i - num\_s + 1);

q = next\_store[q] - 1;

}

}

}