|  |
| --- |
| **北京林业大学ACM爱好者协会** |
| **ACM/ICPC** |
| **大学生程序设计竞赛常用算法汇编** |
|  |
| **作者：徐犇; 金海峰; 王竹** |
| **2012/10/18** |

|  |
| --- |
| ***Copyright © 2009-2012 BJFU ACM Team,All Rights Reserved*** |

# 目录

[目录 I](#_Toc339990166)

[一． 小工具 1](#_Toc339990167)

[C++”万能”头 1](#_Toc339990168)

[Java代码框架 1](#_Toc339990169)

[组合工具 1](#_Toc339990170)

[排列工具 2](#_Toc339990171)

[进制工具 2](#_Toc339990172)

[输入外挂 3](#_Toc339990173)

[C++大数类 4](#_Toc339990174)

[二． 常用定理及公式 8](#_Toc339990175)

[求和公式 8](#_Toc339990176)

[组合公式 8](#_Toc339990177)

[积分表 8](#_Toc339990178)

[巴什博弈（Bash Game） 8](#_Toc339990179)

[威佐夫博弈（Wythoff Game） 9](#_Toc339990180)

[尼姆博弈（Nimm Game） 9](#_Toc339990181)

[费马大定理 9](#_Toc339990182)

[费马小定理 9](#_Toc339990183)

[欧拉公式 9](#_Toc339990184)

[中国剩余定理 9](#_Toc339990185)

[哥德巴赫猜想 9](#_Toc339990186)

[求余运算的性质 9](#_Toc339990187)

[三． 线性代数 9](#_Toc339990188)

[高斯消元法 9](#_Toc339990189)

[矩阵乘法与快速幂 11](#_Toc339990190)

[行列式求值 12](#_Toc339990191)

[四． 数论 13](#_Toc339990192)

[普通法打素数表 13](#_Toc339990193)

[筛法打素数表 13](#_Toc339990194)

[打π表 13](#_Toc339990195)

[暴力生成法里数列 14](#_Toc339990196)

[求n阶法里数列第k项(k<n) 14](#_Toc339990197)

[巧求阶乘位数(斯特林公式) 15](#_Toc339990198)

[大数取模 16](#_Toc339990199)

[快速幂a^b 16](#_Toc339990200)

[乘积取余a\*b%c 16](#_Toc339990201)

[快速幂取余a^b%c 16](#_Toc339990202)

[快速幂取余a^b%c（a,b很大） 16](#_Toc339990203)

[筛选素数 17](#_Toc339990204)

[得到区间素数 17](#_Toc339990205)

[米勒-罗宾素数测试 18](#_Toc339990206)

[大数素数分解 18](#_Toc339990207)

[GCD & LCM Inverse 19](#_Toc339990208)

[反素数zoj2562 timus 1748 19](#_Toc339990209)

[求出N的所有素因子及其个数 20](#_Toc339990210)

[求N的所有约数 20](#_Toc339990211)

[求N的阶乘约数的个数 21](#_Toc339990212)

[最大公约数(GCD)欧几里德算法 21](#_Toc339990213)

[“快速” GCD 21](#_Toc339990214)

[扩展欧几里德 22](#_Toc339990215)

[模线性方程 22](#_Toc339990216)

[递推求欧拉函数值 22](#_Toc339990217)

[单独求欧拉函数值 22](#_Toc339990218)

[利用素数表求欧拉函数值 23](#_Toc339990219)

[求欧拉函数的和 23](#_Toc339990220)

[满足gcd(n,i)=1的个数 23](#_Toc339990221)

[满足a^x=1(mod n)最小的x 24](#_Toc339990222)

[poj 3696 M=mmmm M%k==0 24](#_Toc339990223)

[最小公倍数和 25](#_Toc339990224)

[模线性方程组 26](#_Toc339990225)

[离散对数（discrete logging） 26](#_Toc339990226)

[特殊Baby Step Giant Step 26](#_Toc339990227)

[一般Baby Step Giant Step 26](#_Toc339990228)

[x^2=n (mod m) 是不是二次剩余 28](#_Toc339990229)

[X^k = n mod p (p is prime) 28](#_Toc339990230)

[x^2=1 (mod n)解的个数 29](#_Toc339990231)

[小知识 30](#_Toc339990232)

[幻方构造 30](#_Toc339990233)

[五． 组合数学 30](#_Toc339990234)

[打组合数表 30](#_Toc339990235)

[排列数的末尾非零数 31](#_Toc339990236)

[排列数组合数的末尾非零数 31](#_Toc339990237)

[组合数取模C(n,m)%p 32](#_Toc339990238)

[组合数C(n,m)与因数k 33](#_Toc339990239)

[Catalan Numbers(卡特兰数) 33](#_Toc339990240)

[Stirling (斯特灵数)的应用 35](#_Toc339990241)

[斐波那契数列 37](#_Toc339990242)

[母函数学习 38](#_Toc339990243)

[普通母函数 38](#_Toc339990244)

[指数型母函数 39](#_Toc339990245)

[Polya定理 39](#_Toc339990246)

[dfs实现容斥原理计数 39](#_Toc339990247)

[排列组合生成 40](#_Toc339990248)

[生成gray码 41](#_Toc339990249)

[置换(polya) 41](#_Toc339990250)

[字典序全排列 41](#_Toc339990251)

[字典序组合 41](#_Toc339990252)

[六． 字符串、匹配 41](#_Toc339990253)

[Trie树 41](#_Toc339990254)

[KMP 42](#_Toc339990255)

[KMP实现DFA(单模式串) 42](#_Toc339990256)

[Trie树实现DFA(多模式串) 43](#_Toc339990257)

[AC自动机（DFA） 44](#_Toc339990258)

[Sunday算法 45](#_Toc339990259)

[七． 数据结构 46](#_Toc339990260)

[RMQ 46](#_Toc339990261)

[一维线段树 46](#_Toc339990262)

[快速排序 47](#_Toc339990263)

[平面上矩形面积并 47](#_Toc339990264)

[一维树状数组 48](#_Toc339990265)

[二维树状数组 49](#_Toc339990266)

[八． 动态规划 49](#_Toc339990267)

[最长公共子序列长度 49](#_Toc339990268)

[数位DP:1-n包含49的个数 49](#_Toc339990269)

[九． 网络流 50](#_Toc339990270)

[匈牙利算法(邻接矩阵) 50](#_Toc339990271)

[匈牙利算法(邻接表) 50](#_Toc339990272)

[最小点集覆盖 51](#_Toc339990273)

[KM算法 51](#_Toc339990274)

[最小费用最大流 51](#_Toc339990275)

[dinic最大流 52](#_Toc339990276)

[十． 图论 54](#_Toc339990277)

[邻接表的存储方法 54](#_Toc339990278)

[拓扑排序 54](#_Toc339990279)

[强连通分支 54](#_Toc339990280)

[无向图最小割 55](#_Toc339990281)

[spfa算法 56](#_Toc339990282)

[求割点 56](#_Toc339990283)

[求边双连通分支 57](#_Toc339990284)

[bellman\_ford邻接阵形式 58](#_Toc339990285)

[bellman-ford邻接表形式 58](#_Toc339990286)

[dijkstra邻接表 59](#_Toc339990287)

[djkstra邻接表+堆优化 60](#_Toc339990288)

[dijkstra邻接阵形式? 60](#_Toc339990289)

[floyd邻接矩阵形式 61](#_Toc339990290)

[普里姆（加点法） 61](#_Toc339990291)

[克鲁斯卡尔（加边法） 62](#_Toc339990292)

[2-sat问题 62](#_Toc339990293)

[十一． 计算几何 63](#_Toc339990294)

[几何题注意事项 63](#_Toc339990295)

[常用几何公式 63](#_Toc339990296)

[三角形 63](#_Toc339990297)

[四边形 63](#_Toc339990298)

[圆 63](#_Toc339990299)

[棱柱 63](#_Toc339990300)

[棱锥 64](#_Toc339990301)

[棱台 64](#_Toc339990302)

[圆柱 64](#_Toc339990303)

[圆锥 64](#_Toc339990304)

[圆台 64](#_Toc339990305)

[球 64](#_Toc339990306)

[球台 64](#_Toc339990307)

[球扇形 64](#_Toc339990308)

[几何题公共头 64](#_Toc339990309)

[叉积 64](#_Toc339990310)

[点积 64](#_Toc339990311)

[多边形 64](#_Toc339990312)

[多边形凹凸判定 64](#_Toc339990313)

[判点在凸多边形内或多边形边上 64](#_Toc339990314)

[多边形重心 65](#_Toc339990315)

[多边形切割 66](#_Toc339990316)

[浮点函数 66](#_Toc339990317)

[两点距离 66](#_Toc339990318)

[判三点共线 66](#_Toc339990319)

[判断点和线段关系 66](#_Toc339990320)

[判两点与线段位置关系 66](#_Toc339990321)

[判两直线平行 67](#_Toc339990322)

[判两直线垂直 67](#_Toc339990323)

[判两线段相交 67](#_Toc339990324)

[计算两直线交点 67](#_Toc339990325)

[点到直线上的最近点 67](#_Toc339990326)

[点到直线距离 67](#_Toc339990327)

[点到线段上的最近点 67](#_Toc339990328)

[点到线段距离 68](#_Toc339990329)

[矢量V以P为顶点逆时针旋转angle并放大scale倍 68](#_Toc339990330)

[面积 68](#_Toc339990331)

[公共头 68](#_Toc339990332)

[计算三角形面积 68](#_Toc339990333)

[计算多边形面积 68](#_Toc339990334)

[球面 68](#_Toc339990335)

[公共头 68](#_Toc339990336)

[计算距离,r为球半径 68](#_Toc339990337)

[计算球面距离,r为球半径 68](#_Toc339990338)

[三角形 68](#_Toc339990339)

[公共头 68](#_Toc339990340)

[外心 69](#_Toc339990341)

[内心 69](#_Toc339990342)

[垂心 69](#_Toc339990343)

[重心 69](#_Toc339990344)

[费马点 69](#_Toc339990345)

[三维几何 69](#_Toc339990346)

[公共头 69](#_Toc339990347)

[三维叉积 69](#_Toc339990348)

[三维点积 70](#_Toc339990349)

[矢量差 U - V 70](#_Toc339990350)

[取平面法向量 70](#_Toc339990351)

[两点距离,单参数取向量大小 70](#_Toc339990352)

[向量大小 70](#_Toc339990353)

[判三点共线 70](#_Toc339990354)

[判四点共面 70](#_Toc339990355)

[判点是否在线段上 70](#_Toc339990356)

[判点是否在空间三角形上 70](#_Toc339990357)

[判两点在线段同侧 71](#_Toc339990358)

[判两点在线段异侧 71](#_Toc339990359)

[判两直线平行 71](#_Toc339990360)

[判两平面平行 71](#_Toc339990361)

[判直线与平面平行 71](#_Toc339990362)

[判两直线垂直 71](#_Toc339990363)

[判两平面垂直 71](#_Toc339990364)

[判直线与平面平行 71](#_Toc339990365)

[判两线段相交 71](#_Toc339990366)

[判线段与空间三角形相交 72](#_Toc339990367)

[计算两直线交点 72](#_Toc339990368)

[计算直线与平面交点 72](#_Toc339990369)

[计算两平面交线 72](#_Toc339990370)

[点到直线距离 73](#_Toc339990371)

[点到平面距离 73](#_Toc339990372)

[直线到直线距离 73](#_Toc339990373)

[两直线夹角cos值 73](#_Toc339990374)

[两平面夹角cos值 73](#_Toc339990375)

[直线平面夹角sin值 73](#_Toc339990376)

[构造凸包 73](#_Toc339990377)

[网格 74](#_Toc339990378)

[多边形上的网格点个数 74](#_Toc339990379)

[多边形内的网格点个数 74](#_Toc339990380)

[圆 74](#_Toc339990381)

[判直线和圆相交 74](#_Toc339990382)

[判线段和圆相交 74](#_Toc339990383)

[判圆和圆相交,包括相切 74](#_Toc339990384)

[计算圆上到点p最近点 74](#_Toc339990385)

[计算直线与圆的交点 74](#_Toc339990386)

[计算圆与圆的交点 75](#_Toc339990387)

[整数函数 75](#_Toc339990388)

[公共头 75](#_Toc339990389)

[判三点共线 75](#_Toc339990390)

[判点是否在线段上 75](#_Toc339990391)

[判两点在直线同侧 75](#_Toc339990392)

[判两点在直线异侧 75](#_Toc339990393)

[判两直线平行 75](#_Toc339990394)

[判两直线垂直 75](#_Toc339990395)

[判两线段相交 76](#_Toc339990396)

[旋转卡壳法求凸包直径 76](#_Toc339990397)

[是否是凸包（要求逆时针） 76](#_Toc339990398)

[点是否在凸包内（要求逆时针） 76](#_Toc339990399)

[欧拉回路（无向图） 76](#_Toc339990400)

[点在多边形内（转角法） 77](#_Toc339990401)

[Pick公式 77](#_Toc339990402)

[极坐标排序与二分查找 78](#_Toc339990403)

[十二． 其它 78](#_Toc339990404)

[高精度样例-大数阶乘函数 78](#_Toc339990405)

[递归下降法样例-表达式求值 79](#_Toc339990406)

[若干STL数据结构用法 80](#_Toc339990407)

[Java正则表达式用法样例 80](#_Toc339990408)

[附录A. 国内外著名OJ简介 80](#_Toc339990409)

[北大OJ 80](#_Toc339990410)

[杭电OJ 80](#_Toc339990411)

[附录B. GNU C与ANSI C的区别 80](#_Toc339990412)

1. 小工具

C++”万能”头

/\*

\* ${project\_name}/${file\_name}

\* Created on: ${date}

\* Author : ben

\*/

**#**include<cstdio>

#include <cstdlib>

#include <cstring>

#include <cmath>

#include <ctime>

#include <iostream>

#include <algorithm>

#include <queue>

#include <set>

#include <map>

#include <stack>

#include <string>

#include <vector>

#include <deque>

#include <list>

#include <functional>

#include <numeric>

#include <cctype>

using namespace std;

typedef long long LL;

int main() {

#ifndef ONLINE\_JUDGE

freopen("data.in", "r", stdin);

#endif

return 0;

}

Java代码框架

import java.io.\*;

import java.util.\*;

public class Main {

public static Scanner getFileScanner(boolean isOnlineJudge) {

if(isOnlineJudge) {

return new Scanner(System.in);

}

File myFile = new File("data.in");

FileInputStream myFileStream = null;

try {

myFileStream = new FileInputStream(myFile);

} catch (FileNotFoundException e) {

}

Scanner cin = new Scanner(myFileStream);

return cin;

}

public static void main(String[] args) {

//获取输入对象，参数为true时为标准输入，否则从data.in文件读入

Scanner cin = getFileScanner(false);

}

}

组合工具

/\*\*

\* 用于穷举过程中按顺序生成所有的组合，用法示例如下：

CCombination c;

c.init(5, 3);

while (c.next()) {

cout << c.data[0] << c.data[1] << c.data[2] << endl;

}

\* c.init(5, 3)即为将组合对象初始化为5个中选3个

\* 循环语句就可以输出所有的组合情况如012、013、234等

\*/

class CCombination {

public:

int total;

int elite;

int\* data;

void init(int givenTotal, int givenElite);

bool next();

~CCombination();

};

CCombination::~CCombination() {

if (data != NULL)

delete[] data;

}

void CCombination::init(int givenTotal, int givenElite) {

total = givenTotal;

elite = givenElite;

data = new int[total];

data[0] = -1; //表示下一个才是开始，是第一个

}

bool CCombination::next() {

int i = 0;

if (data[0] == -1) {

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

data[i] = i;

return true;

}

int pos = -1;

for (i = elite - 1; i >= 0; i--) {

if (data[i] < total - elite + i) {

pos = i;

break;

}

}

if (pos == -1)

return false;

data[pos]++;

for (i = pos + 1; i < elite; i++)

data[i] = data[pos] + (i - pos);

return true;

}

排列工具

/\*\*

\* 与上面组合工具用法类似，只是生成的为所有排列

\* 如012、013、210、432等

\*/

class CPermutation {

public:

int total;

int elite;

int \*data;

void init(int givenTotal, int givenElite);

bool next();

~CPermutation();

};

CPermutation::~CPermutation() {

if (data != NULL)

delete[] data;

}

void CPermutation::init(int givenTotal, int givenElite) {

total = givenTotal;

elite = givenElite;

data = new int[total];

data[0] = -1;

}

bool CPermutation::next() {

int i = 0, j = 0, k = 0, max;

bool t;

if (data[0] == -1) {

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

data[i] = i;

return true;

}

int pos = -1;

for (i = elite - 1; i >= 0; i--) {

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

t = true;

for (k = 0; k < i; k++) {

if (data[k] == j) {

t = false;

break;

}

}

if (t) {

max = j;

break;

}

}

if (data[i] < max) {

pos = i;

break;

}

}

if (pos == -1)

return false;

for (j = data[pos] + 1; j < total; j++) {

t = true;

for (k = 0; k < pos; k++) {

if (j == data[k]) {

t = false;

break;

}

}

if (t) {

data[pos] = j;

break;

}

}

for (i = pos + 1; i < elite; i++) {

for (j = 0; j < total; j++) {

t = true;

for (k = 0; k < i; k++) {

if (j == data[k]) {

t = false;

break;

}

}

if (t) {

data[i] = j;

break;

}

}

}

return true;

}

进制工具

/\*\*

\* 与上面组合工具用法类似，只是生成的序列

\* 如011、012、444等

\*/

class CDigital {

public:

int total;

int elite;

int\* data;

void init(int givenTotal, int givenElite);

bool next();

~CDigital();

};

CDigital::~CDigital() {

if (data != NULL)

delete[] data;

}

void CDigital::init(int givenTotal, int givenElite) {

total = givenTotal;

elite = givenElite;

data = new int[total];

data[0] = -1; //表示下一个才是开始，是第一个

}

bool CDigital::next() {

int i = 0;

if (data[0] == -1) {

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

data[i] = 0;

return true;

}

int pos = -1;

for (i = elite - 1; i >= 0; i--) {

if (data[i] < total - 1) {

pos = i;

break;

}

}

if (pos == -1)

return false;

data[pos]++;

for (i = pos + 1; i < elite; i++)

data[i] = 0;

return true;

}

输入外挂

/\*\*

\* 在ACM竞赛中，核心的是算法的优化。但有的时候，一些别的

\* 的细节，也会对是否通过题目有所影响。程序的输入输出

\* 方法即是一例。通常，使用scanf函数会比使用cin输入快很

\* 多，但是使用下面的输入外挂，将会比scanf函数更快，原因

\* 是scanf中有许多额外的、关于健壮性的处理和判断。作者在

\* 多台pc机上，g++3.4.5环境中进行对比测试，发现以下函数

\* 比scanf快了5~10倍，比cin快了近100倍。

\*/

//输入非负整数，用法int a = get\_int();

int get\_int() {

int res = 0, ch;

while (!((ch = getchar()) >= '0' && ch <= '9')) {

if (ch == EOF)

return 1 << 30;

}

res = ch - '0';

while ((ch = getchar()) >= '0' && ch <= '9')

res = res \* 10 + (ch - '0');

return res;

}

//输入整数(包括负整数)，用法int a = get\_int2();

int get\_int2() {

int res = 0, ch, flag = 0;

while (!((ch = getchar()) >= '0' && ch <= '9')) {

if (ch == '-')

flag = 1;

if (ch == EOF)

return 1 << 30;

}

res = ch - '0';

while ((ch = getchar()) >= '0' && ch <= '9')

res = res \* 10 + (ch - '0');

if (flag == 1)

res = -res;

return res;

}

//输入字符串(串中不包含空格等)，用法get\_str(str);

bool get\_str(char \*str) {

char c;

while ((c = getchar()) <= ' ') {

if(c == EOF) {

return false;

}

}

int I = 0;

while (c > ' ') {

str[I++] = c; c = getchar();

}

str[I] = 0;

return true;

}

C++大数类

#include <iostream>

#include <string>

using namespace std;

class BigInteger {

public:

BigInteger() {

}

BigInteger(char s[]);

BigInteger(int i);

public:

//重载赋值操作，允许用普通数给大数赋值

int operator=(int bn2);

//重载赋值操作，允许用字符串形式的大数给其赋值

char \* operator=(char \*s1);

public:

//重载输入操作符

friend istream& operator >>(istream& input, BigInteger& bn) {

input >> bn.m\_s;

return input;

}

//重载输出操作符

friend ostream& operator <<(ostream& output, BigInteger& bn) {

output << bn.m\_s;

return output;

}

friend BigInteger operator +(BigInteger bn1, BigInteger bn2);

friend BigInteger operator -(BigInteger bn1, BigInteger bn2);

friend BigInteger operator -(BigInteger bn1);

friend BigInteger operator \*(BigInteger bn1, BigInteger bn2);

friend BigInteger operator %(BigInteger bn1, BigInteger bn2);

friend int operator %(BigInteger bn1, int bn2);

friend BigInteger operator /(BigInteger bn1, BigInteger bn2);

friend bool operator <(const BigInteger& bn1, const BigInteger& bn2);

friend bool operator <=(const BigInteger& bn1, const BigInteger& bn2);

friend bool operator >(const BigInteger& bn1, const BigInteger& bn2);

friend bool operator >(const BigInteger& bn1, int bn2);

friend bool operator >=(const BigInteger& bn1, const BigInteger& bn2);

friend bool operator ==(const BigInteger& bn1, const BigInteger& bn2);

friend bool operator ==(const BigInteger& bn1, int bn2);

private:

//获得大数的位数

int size() const {

return m\_s.size();

}

private:

//内部用string来存储大数

string m\_s;

};

//实现用普通数构造大数

BigInteger::BigInteger(int i) {

char a[2] = { 0 };

if (i == 0) {

m\_s = "0";

return;

}

for (int tmp; i > 0; i /= 10) {

tmp = i % 10;

a[0] = tmp + '0';

m\_s = a + m\_s;

}

}

//实现用字符串构造大数

BigInteger::BigInteger(char s[]) :

m\_s(s) {

}

//实现重载赋值操作，允许用普通数给大数赋值

int BigInteger::operator=(int bn2) {

char a[2] = { 0 };

m\_s = "";

if (bn2 == 0) {

m\_s = "0";

return bn2;

}

for (int tmp; bn2 > 0; bn2 /= 10) {

tmp = bn2 % 10;

a[0] = tmp + '0';

m\_s = a + m\_s;

}

return bn2;

}

//实现重载赋值操作，允许用字符串形式的大数给其赋值

char \* BigInteger::operator=(char \*s1) {

m\_s = s1;

return s1;

}

//实现小于操作重载

bool operator<(const BigInteger& bn1, const BigInteger& bn2) {

if (bn2.m\_s.length() == 1 && bn2.m\_s[0] == '0') {

if (bn1.m\_s[0] == '-') {

return true;

} else {

return false;

}

}

if (bn1.size() < bn2.size()) {

return true;

} else if (bn1.size() > bn2.size()) {

return false;

}

for (int i = 0, length = bn1.size(); i < length; ++i) {

if (bn1.m\_s[i] < bn2.m\_s[i]) {

return true;

} else if (bn1.m\_s[i] > bn2.m\_s[i]) {

return false;

}

}

return false;

}

//实现等于操作重载

bool operator==(const BigInteger& bn1, const BigInteger& bn2) {

if (bn1.size() != bn2.size()) {

return false;

}

for (int i = 0, length = bn1.size(); i < length; ++i) {

if (bn1.m\_s[i] != bn2.m\_s[i]) {

return false;

}

}

return true;

}

//实现小于等于重载

bool operator<=(const BigInteger& bn1, const BigInteger& bn2) {

if (bn1 < bn2 || bn1 == bn2) {

return true;

} else {

return false;

}

}

//使大数可以跟普通数比较

bool operator==(const BigInteger& bn1, int bn2) {

BigInteger i(bn2);

if (bn1 == i) {

return true;

}

return false;

}

bool operator>(const BigInteger& bn1, const BigInteger& bn2) {

if (bn1 < bn2 == false && (bn1 == bn2) == false) {

return true;

} else {

return false;

}

}

bool operator>(const BigInteger& bn1, int bn2) {

BigInteger b(bn2);

if (bn1 > b) {

return true;

} else {

return false;

}

}

bool operator>=(const BigInteger& bn1, const BigInteger& bn2) {

if (bn1 > bn2 || bn1 == bn2) {

return true;

} else {

return false;

}

}

BigInteger operator+(BigInteger bn1, BigInteger bn2) {

BigInteger sum;

char a[2];

a[0] = '0';

a[1] = 0;

if (bn1 == 0) {

return bn2;

} else if (bn2 == 0) {

return bn1;

} else if (bn1 > 0 && bn2 < 0) {

bn2 = -bn2;

return bn1 - bn2;

} else if (bn1 < 0 && bn2 > 0) {

bn1 = -bn1;

return bn2 - bn1;

} else if (bn1 < 0 && bn2 < 0) {

bn1 = -bn1;

bn2 = -bn2;

sum = bn1 + bn2;

return -sum;

}

for (int i = bn1.m\_s.size() - 1, j = bn2.m\_s.size() - 1; i >= 0 || j >= 0;--i, --j) {

int tmp;

if (i < 0) {

tmp = bn2.m\_s[j] - '0' + a[0] - '0';

} else if (j < 0) {

tmp = bn1.m\_s[i] - '0' + a[0] - '0';

} else {

tmp = bn1.m\_s[i] - '0' + bn2.m\_s[j] - '0' + a[0] - '0';

}

a[0] = tmp % 10 + '0';

sum.m\_s = a + sum.m\_s;

a[0] = tmp / 10 + '0';

}

if (a[0] != '0') {

sum.m\_s = a + sum.m\_s;

}

return sum;

}

BigInteger operator-(BigInteger bn1) {

if (bn1 > 0)

bn1.m\_s = '-' + bn1.m\_s;

else {

bn1.m\_s = bn1.m\_s.substr(1);

}

return bn1;

}

BigInteger operator-(BigInteger bn1, BigInteger bn2) {

BigInteger sum;

char a[2];

a[0] = '0';

a[1] = 0;

if (bn1 == 0) {

return -bn2;

} else if (bn2 == 0) {

return bn1;

} else if (bn1 > 0 && bn2 < 0) {

bn2 = -bn2;

return bn1 + bn2;

} else if (bn1 < 0 && bn2 > 0) {

bn1 = -bn1;

sum = bn2 + bn1;

return -sum;

} else if (bn1 < 0 && bn2 < 0) {

bn1 = -bn1;

bn2 = -bn2;

return bn2 - bn1;

}

if (bn1 >= bn2) {

for (int i = bn1.m\_s.size() - 1, j = bn2.m\_s.size() - 1;

i >= 0 || j >= 0; --i, --j) {

int tmp;

if (i < 0) {

tmp = bn2.m\_s[j] + a[0] - 96;

} else if (j < 0) {

tmp = bn1.m\_s[i] + a[0] - 96;

} else {

tmp = bn1.m\_s[i] - bn2.m\_s[j] + a[0] - '0';

}

if (tmp < 0) {

a[0] = 10 + tmp + '0';

sum.m\_s = a + sum.m\_s;

a[0] = 47;

} else {

a[0] = tmp + '0';

sum.m\_s = a + sum.m\_s;

a[0] = 48;

}

}

unsigned ii = 0, ll = sum.m\_s.length();

while (sum.m\_s[ii] == '0') {

ii++;

}

if (ii == 0) {

return sum;

} else if (ii == ll) {

sum.m\_s = "0";

return sum;

} else {

sum.m\_s = sum.m\_s.substr(ii, ll - ii);

return sum;

}

} else {

for (int i = bn2.m\_s.size() - 1, j = bn1.m\_s.size() - 1;

i >= 0 || j >= 0; --i, --j) {

int tmp;

if (i < 0) {

tmp = bn1.m\_s[j] + a[0] - 96;

} else if (j < 0) {

tmp = bn2.m\_s[i] + a[0] - 96;

} else {

tmp = bn2.m\_s[i] - bn1.m\_s[j] + a[0] - '0';

}

if (tmp < 0) {

a[0] = 10 + tmp + '0';

sum.m\_s = a + sum.m\_s;

a[0] = 47;

} else {

a[0] = tmp + '0';

sum.m\_s = a + sum.m\_s;

a[0] = 48;

}

}

unsigned ii = 0, ll = sum.m\_s.length();

while (sum.m\_s[ii] == '0')

ii++;

if (ii != 0) {

sum.m\_s = sum.m\_s.substr(ii, ll - ii);

}

a[0] = '-';

sum.m\_s = a + sum.m\_s;

return sum;

}

}

//大数相乘

BigInteger operator\*(BigInteger bn1, BigInteger bn2) {

BigInteger sum;

if (bn1 < 0 && bn2 > 0) {

bn1 = -bn1;

sum = bn1 \* bn2;

return -sum;

} else if (bn1 > 0 && bn2 < 0) {

bn2 = -bn2;

sum = bn1 \* bn2;

return -sum;

} else if (bn1 < 0 && bn2 < 0) {

bn1 = -bn1;

bn2 = -bn2;

return bn1 \* bn2;

}

int i, c = 0, N, temp;

int m = bn1.m\_s.size() - 1;

int n = bn2.m\_s.size() - 1;

int L = m + n + 2;

char \*a = new char[L + 1];for

( i = 0; i < L; i++) {

a[i] = '0';

}

a[L] = '\0';

while (n >= 0) {

i = m;

N = bn2.m\_s[n] - 48;

while (i >= 0) {

c += N \* (bn1.m\_s[i] - 48);

temp = c % 10;

a[i + n + 1] += temp;

if (a[i + n + 1] > 57) {

a[i + n + 1] -= 10;

a[i + n]++;

}

c /= 10;

i--;

}

if (c != 0) {

a[i + n + 1] += c;

if (a[i + n + 1] > 57) {

a[i + n + 1] -= 10;

a[i + n]++;

}

c = 0;

}

n--;

}

while (a[0] == '0') {

a++;

}

sum = a;

return sum;

}

//大数对大数取余(未实现)

BigInteger operator%(BigInteger bn1, BigInteger bn2) {

BigInteger sum("0");

return sum;

}

//大数对整数取余

int operator%(BigInteger bn1, int bn2) {

int j = 0, i, l = bn1.m\_s.size();

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

j = (j \* 10 % bn2 + (bn1.m\_s[i] - '0') % bn2) % bn2;

}

return j;

}

//大数除大数

BigInteger operator/(BigInteger bn1, BigInteger bn2) {

BigInteger c, sum;

if (bn1 < 0 && bn2 > 0) {

bn1 = -bn1;

sum = bn1 / bn2;

return -sum;

} else if (bn1 > 0 && bn2 < 0) {

bn2 = -bn2;

sum = bn1 / bn2;

return -sum;

} else if (bn1 < 0 && bn2 < 0) {

bn1 = -bn1;

bn2 = -bn2;

return bn1 / bn2;

}

if (bn1 < bn2) {

return 0;

}

char \*a;

unsigned l1 = bn1.m\_s.length(), l2 = bn2.m\_s.length();

int L = l1 - l2 + 1, I, i;

a = new char[L + 1];for

( i = 0; i < L; i++) {

a[i] = '0';}

a[L] = '\0';

c.m\_s.assign(bn1.m\_s, 0, l2);

for (I = 0; I < L; I++) {

while (c >= bn2) {

c = c - bn2;

a[I]++;

}

c.m\_s = c.m\_s + bn1.m\_s[I + l2];

}

while (a[0] == '0') {

a++;

}

sum = a;

return sum;

}

int main() {

BigInteger a, b, c;

while (cin >> a >> b) {

c = a \* b;

cout << c << endl;

}

return 0;

}

1. 常用定理及公式

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

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

组合公式

1.

2.

derangement D(n) = n!(1 - 1/1! + 1/2! - 1/3! + ... + (-1)^n/n!)

= (n-1)(D(n-2) - D(n-1))

Q(n) = D(n) + D(n-1)

积分表

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

11.

12.

13.

14.

15.

16.

17.

18.

19.

20.

21. + C

22.

23.

24.

巴什博弈（Bash Game）

只有一堆n个物品，两个人轮流从这堆物品中取物，规定每次至少取一个，最多取 m 个。最后取光者得胜。

显而易见，当剩余的石子个数为 1，2...m 时，此状态必为必胜状态，由此可知，当剩余的石子个数为 m+1时为必败状态，但是当剩余石子个数为 m+2, m+3 ... 2m+1 时，玩家可以取走一定量的石子，使剩余的石子数为必败状态的 m+1 个石子，故当剩余石子个数为 m+2, m+3 ... 2m+1 时为必胜状态。以此类推，可知当石子数为 k\*( m+1 ) 时为必败状态。

威佐夫博弈（Wythoff Game）

有两堆各若干个物品，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。首先，我们知道（0，0）是必败态，再仔细往下推就可以知道（1，2）、（3，5）、（4，7）、（6，10）等都是必败态，除了必败态之外就是必胜态。具体原因读者自己想想就能明白。那么如何判断一个局势是否必败态呢？公式可能有很多，但最经典的是其通项公式，即ak =[k（1+√5）/2]，bk= ak + k （k=0，1，2，...,n 方括号表示取整函数)。奇妙的是其中出现了黄金分割数（1+√5）/2 = 1.618。

尼姆博弈（Nimm Game）

有三堆各若干个物品，两个人轮流从某一堆取任意多的物品，规定每次至少取一个，多者不限，最后取光者得胜。

首先（0，0，0）显然是奇异局势，无论谁面对奇异局势，都必然失败。第二种奇异局势是（0，n，n），只要与对手拿走一样多的物品，最后都将导致（0，0，0）。

对于任何奇异局势(a,b,c)，都有a^b^c=0.

非奇异局势(a,b,c)(a<b<c)转换为奇异局势，只需将c变为a^b，即从c中减去 c-(a^b)即可。

当然，该博弈也可推广到多堆石子，同样也可以是两堆石子（要注意此时它与威佐夫博弈的区别）

费马大定理

不可能将一个3次方分成两个3次方之和；一个4次方不可能写成两个4次方之和；一般地，任何高于2次的幂不可能写成两个同次幂之和。

费马小定理

设p是素数，a是任意整数且a0(mod p)，则ap-1≡1(mod p)。

欧拉公式

如果gcd(a, b) = 1，则aφ(m) ≡ 1(mod m)。

其中φ(m)为欧拉函数，表示0到m之间与m互素的整数个数。

中国剩余定理

设m与n是整数，gcd(m, n) = 1，b与c是任意整数。则同余式组

x ≡ b(mod m) 与 x ≡ c(mod n)恰有一个解0≤x≤mn。

哥德巴赫猜想

每个偶数n≥4可表成两个素数之和

求余运算的性质

基本性质：

①若p | (a - b)，则a ≡ b (% p)。

②若(a % p) = (b % p)，则a≡b (% p)。

③对称性：若a ≡ b (% p)，则b ≡ a (% p)。

④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：

①(a + b) % p = (a % p + b % p) % p

②(a - b) % p = (a % p - b % p) % p

③(a \* b) % p = (a % p \* b % p) % p

④ab % p = ((a % p)b) % p

⑤结合率：

((a + b) % p + c) % p = (a + (b + c) % p) % p

((a \* b) % p \* c) % p = (a \* (b \* c) % p) % p

⑥交换率：

(a + b) % p = (b + a) % p

(a \* b) % p = (b \* a) % p

⑦分配率：

((a + b) % p \* c) % p = ((a \* c) % p + (b \* c) % p) % p

其它性质：

①若a ≡ b (% p), 则对于任意的c, 都有(a + c) ≡ (b + c) (% p)

②若a ≡ b (% p), 则对于任意的c, 都有(a - c) ≡ (b - c) (% p)

③若a ≡ b (% p), 则对于任意的c，都有(a \* c) ≡ (b \* c) (% p)

④若a ≡ b (% p), 则对于任意的c，都有ac ≡ bc (% p)

⑤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)

⑥如果a \* c ≡ b \* c (% p)，且c与p互质，则a ≡ b (% p）

1. 线性代数

高斯消元法

是线性代数中的一个算法，可用来求解线性方程组，并可以求出矩阵的秩，以及求出可逆方阵的逆矩阵。高斯消元法的原理是：

若用初等行变换将增广矩阵 化为 ，则AX = B与CX = D是同解方程组。

所以我们可以用初等行变换把增广矩阵转换为行阶梯阵，然后回代求出方程的解。

以上是线性代数课的回顾，下面来说说高斯消元法在编程中的应用。

首先，先介绍程序中高斯消元法的步骤：

(我们设方程组中方程的个数为equ，变元的个数为var，注意：一般情况下是n个方程，n个变元，但是有些题目就故意让方程数与变元数不同)

1. 把方程组转换成增广矩阵。

2. 利用初等行变换来把增广矩阵转换成行阶梯阵。

枚举k从0到equ – 1，当前处理的列为col(初始为0) ，每次找第k行以下(包括第k行)，col列中元素绝对值最大的列与第k行交换。如果col列中的元素全为0，那么则处理col + 1列，k不变。

3. 转换为行阶梯阵，判断解的情况。

① 无解

当方程中出现(0, 0, …, 0, a)的形式，且a != 0时，说明是无解的。

② 唯一解

条件是k = equ，即行阶梯阵形成了严格的上三角阵。利用回代逐一求出解集。

③ 无穷解。

条件是k < equ，即不能形成严格的上三角形，自由变元的个数即为equ – k，但有些题目要求判断哪些变元是不缺定的。

    这里单独介绍下这种解法：

首先，自由变元有var - k个，即不确定的变元至少有var - k个。我们先把所有的变元视为不确定的。在每个方程中判断不确定变元的个数，如果大于1个，则该方程无法求解。如果只有1个变元，那么该变元即可求出，即为确定变元。

以上介绍的是求解整数线性方程组的求法，复杂度是O(n3)。浮点数线性方程组的求法类似，但是要在判断是否为0时，加入EPS，以消除精度问题。

这里提供下自己写的还算满意的求解整数线性方程组的模板(浮点数类似，就不提供了)～～

/\* 用于求整数解得方程组. \*/

const int maxn = 105;

int equ, var; // 有equ个方程，var个变元。增广阵行数为equ, 分别为0到equ - 1，列数为var + 1，分别为0到var.

int a[maxn][maxn];

int x[maxn]; // 解集.

bool free\_x[maxn]; // 判断是否是不确定的变元.

int free\_num;

void Debug(void) {

int i, j;

for (i = 0; i < equ; i++) {

for (j = 0; j < var + 1; j++) {

cout << a[i][j] << " ";

}

cout << endl;

}

cout << endl;

}

inline int gcd(int a, int b) {

int t;

while (b != 0) {

t = b;

b = a % b;

a = t;

}

return a;

}

inline int lcm(int a, int b) {

return a \* b / gcd(a, b);

}

// 高斯消元法解方程组(Gauss-Jordan elimination).(-2表示有浮点数解，但无整数解，-1表示无解，0表示唯一解，大于0表示无穷解，并返回自由变元的个数)

int Gauss(void) {

int i, j, k;

int max\_r; // 当前这列绝对值最大的行.

int col; // 当前处理的列.

int ta, tb;

int LCM;

int temp;

int free\_x\_num;

int free\_index;

// 转换为阶梯阵.

col = 0; // 当前处理的列.

for (k = 0; k < equ && col < var; k++, col++) { // 枚举当前处理的行.

// 找到该col列元素绝对值最大的那行与第k行交换.(为了在除法时减小误差)

max\_r = k;

for (i = k + 1; i < equ; i++) {

if (abs(a[i][col]) > abs(a[max\_r][col]))

max\_r = i;

}

if (max\_r != k) { // 与第k行交换.

for (j = k; j < var + 1; j++)

swap(a[k][j], a[max\_r][j]);

}

if (a[k][col] == 0) { // 说明该col列第k行以下全是0了，则处理当前行的下一列.

k--;

continue;

}

for (i = k + 1; i < equ; i++) { // 枚举要删去的行.

if (a[i][col] != 0) {

LCM = lcm(abs(a[i][col]), abs(a[k][col]));

ta = LCM / abs(a[i][col]), tb = LCM / abs(a[k][col]);

if (a[i][col] \* a[k][col] < 0)

tb = -tb; // 异号的情况是两个数相加.

for (j = col; j < var + 1; j++) {

a[i][j] = a[i][j] \* ta - a[k][j] \* tb;

}

}

}

}

Debug();

// 1. 无解的情况: 化简的增广阵中存在(0, 0, ..., a)这样的行(a != 0).

for (i = k; i < equ; i++) { // 对于无穷解来说，如果要判断哪些是自由变元，那么初等行变换中的交换就会影响，则要记录交换.

if (a[i][col] != 0)

return -1;

}

// 2. 无穷解的情况: 在var \* (var + 1)的增广阵中出现(0, 0, ..., 0)这样的行，即说明没有形成严格的上三角阵.

// 且出现的行数即为自由变元的个数.

if (k < var) {

// 首先，自由变元有var - k个，即不确定的变元至少有var - k个.

for (i = k - 1; i >= 0; i--) {

// 第i行一定不会是(0, 0, ..., 0)的情况，因为这样的行是在第k行到第equ行.

// 同样，第i行一定不会是(0, 0, ..., a), a != 0的情况，这样的无解的.

free\_x\_num = 0; // 用于判断该行中的不确定的变元的个数，如果超过1个，则无法求解，它们仍然为不确定的变元.

for (j = 0; j < var; j++) {

if (a[i][j] != 0 && free\_x[j])

free\_x\_num++, free\_index = j;

}

if (free\_x\_num > 1)

continue; // 无法求解出确定的变元.

// 说明就只有一个不确定的变元free\_index，那么可以求解出该变元，且该变元是确定的.

temp = a[i][var];

for (j = 0; j < var; j++) {

if (a[i][j] != 0 && j != free\_index)

temp -= a[i][j] \* x[j];

}

x[free\_index] = temp / a[i][free\_index]; // 求出该变元.

free\_x[free\_index] = 0; // 该变元是确定的.

}

return var - k; // 自由变元有var - k个.

}

// 3. 唯一解的情况: 在var \* (var + 1)的增广阵中形成严格的上三角阵.

// 计算出Xn-1, Xn-2 ... X0.

for (i = var - 1; i >= 0; i--) {

temp = a[i][var];

for (j = i + 1; j < var; j++) {

if (a[i][j] != 0)

temp -= a[i][j] \* x[j];

}

if (temp % a[i][i] != 0)

return -2; // 说明有浮点数解，但无整数解.

x[i] = temp / a[i][i];

}

return 0;

}

int main(void) {

freopen("Input.txt", "r", stdin);

int i, j;

while (scanf("%d %d", &equ, &var) != EOF) {

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

memset(x, 0, sizeof(x));

memset(free\_x, 1, sizeof(free\_x)); // 一开始全是不确定的变元.

for (i = 0; i < equ; i++) {

for (j = 0; j < var + 1; j++) {

scanf("%d", &a[i][j]);

}

}

// Debug();

free\_num = Gauss();

if (free\_num == -1)

printf("无解!\n");

else if (free\_num == -2)

printf("有浮点数解，无整数解!\n");

else if (free\_num > 0) {

printf("无穷多解! 自由变元个数为%d\n", free\_num);

for (i = 0; i < var; i++) {

if (free\_x[i])

printf("x%d 是不确定的\n", i + 1);

else

printf("x%d: %d\n", i + 1, x[i]);

}

} else {

for (i = 0; i < var; i++) {

printf("x%d: %d\n", i + 1, x[i]);

}

}

printf("\n");

}

return 0;

}

矩阵乘法与快速幂

// 比较通用的矩阵，最大阶数由MAX\_ORDER决定，根据需要可以直接修改。row为行数, col为列数。矩阵乘法题的题目一般都会要求对某个数取余，如果这个数固定，可以直接修改MOD的值，也可以将MOD前const去掉以便动态地修改，如果不需要取余，可以将MOD改为1或者去掉

const int MAX\_ORDER = 105;

const int MOD = 2;

typedef short int typec;

typedef struct MyMatrix {

int row, col;

typec num[MAX\_ORDER][MAX\_ORDER];

MyMatrix(int rr, int cc) {

row = rr;

col = cc;

}

inline void init() {

memset(num, 0, sizeof(num));

}

} MyMatrix;

//矩阵乘法。注意：ma.col与mb.row一定要相等，否则会出问题

MyMatrix operator\*(MyMatrix ma, MyMatrix mb) {

int row = ma.row;

int col = mb.col;

int K = ma.col;

MyMatrix numc(row, col);

numc.init();

int i, j, k;

for (i = 0; i < row; i++) {

for (j = 0; j < col; j++) {

for (k = 0; k < K; k++) {

numc.num[i][j] += ma.num[i][k] \* mb.num[k][j];

numc.num[i][j] %= MOD;

}

}

}

return numc;

}

//矩阵快速幂。注意：ma.col与ma.row一定要相等，否则会出问题

MyMatrix mpow(MyMatrix ma, int x) {

int ord = ma.row;

MyMatrix numc(ord, ord);

numc.init();

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

numc.num[i][i] = 1;

}

for (; x; x >>= 1) {

if (x & 1) {

numc = numc \* ma;

}

ma = ma \* ma;

}

return numc;

}

行列式求值

import java.io.BufferedInputStream;

import java.math.BigDecimal;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner cin = new Scanner(new BufferedInputStream(System.*in*));

int n, i, j;

BigDecimal[][] num;

while (cin.hasNext()) {

n = cin.nextInt();

num = new BigDecimal[n + 1][n + 1];

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

for (j = 0; j < n; j++) {

num[i][j] = cin.nextBigDecimal();

}

}

if (n == 1) {

System.*out*.println(num[0][0].setScale(2));

continue;

}

System.*out*.println(*solve*(num, n).setScale(2));

}

}

public static BigDecimal solve(BigDecimal[][] num, int n) {

BigDecimal[][] te = new BigDecimal[n + 1][n + 1];

BigDecimal res, temp;

res = BigDecimal.*ZERO*;

int i, j, k;

if (n > 2) {

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

for (j = 0; j < n - 1; j++) {

for (k = 0; k < n - 1; k++) {

if (k >= i) {

te[j][k] = num[j + 1][k + 1];

} else {

te[j][k] = num[j + 1][k];

}

}

}

temp = num[0][i].multiply(*solve*(te, n - 1));

if ((i & 1) == 1) {

temp = temp.multiply(BigDecimal.*valueOf*(-1.0));

}

res = res.add(temp);

}

} else {

res = num[0][0].multiply(num[1][1]).subtract(

num[0][1].multiply(num[1][0]));

}

return res;

}

}

1. 数论

普通法打素数表

//定义一个布尔数组isPrime，先全部初始化为true，然后执行init函数，isPrime[i]为true表示i为素数，可以再从此数组中输出指定数量的素数。复杂度为O(N1.5)

const int N = 1000000;

bool isPrime[N + 1];

void init\_prime\_table() {

int i, j, s;

for (i = 3; i <= N; i++) {

s = (int) sqrt(i);

for (j = 2; j <= s; j++) {

if (i % j == 0) break;

}

if (j <= s) {

isPrime[i] = false;

}

}

}

//下面再提供一个封装更好的版本，得到所有不超过N的素数表，//存于容器pt中

void get\_prime\_table(int N, vector<int> &pt) {

vector<bool> ip;

ip.resize(N + 1);

fill(ip.begin(), ip.end(), true);

int i, j, s, t = N - 1;

for (i = 3; i <= N; i++) {

s = (int) sqrt(i);

for (j = 2; j <= s; j++) {

if (i % j == 0) break;

}

if (j <= s) { ip[i] = false; t--; }

}

pt.resize(t);

t = 0;

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

if(ip[i]) { pt[t++] = i; }

}

}

筛法打素数表

//如果需要打的素数表太大，则应该使用筛法，使用方法同上一条

const int N = 10000000;

bool isPrime[N + 3];//多用两个元素以免判断边界

void init\_prime\_table() {

int p = 2, q, del;

double temp;

while (p <= N) {

while (!isPrime[p]) { p++; }

if (p > N) {//已经结束

break; }

temp = (double) p;

temp \*= p;

if (temp > N)

break;

while (temp <= N) {

del = (int) temp; isPrime[del] = false;

temp \*= p; }

q = p + 1;

while (q < N) {

while (!isPrime[q]) { q++; }

if (q >= N) { break;}

temp = (double) p;

temp \*= q;

if (temp > N) break;

while (temp <= N) {

del = (int) temp;

isPrime[del] = false;

temp \*= p;

}

q++;

}

p++;

}

}

打π表

//本程序是基于公式π/2=1+1!/3!!+2!/5!!+3!/7!!+...+k!/(2\*k+1)!!+...计算得到，保证精度(最后一位为四舍五入得到)。BIT\_NUM即为打出的π的位数(包括小数点前的3)，修改这个常数(但必须是4的倍数)就可以打出不同长度的π

inline int formula(int i) {

return 2 \* i - 1;

}

void pitable() {

const int BIT\_NUM = 20;

const int SCALE = 10000;

const int BUF\_LEN = BIT\_NUM / 2 \* 7 + 1;

int buf[BUF\_LEN];

int len = BUF\_LEN - 1;

int last = 0;

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

buf[i] = SCALE / 5;

}

while (len > 0) {

int index = len;

int t = buf[index] \* SCALE;

buf[index] = t % formula(index);

t = t / formula(index);

while (--index > 0) {

t = t \* index + buf[index] \* SCALE;

buf[index] = t % formula(index);

t = t / formula(index);

}

len -= 14;

printf("%.4d", last + t / SCALE);

last = t % SCALE;

}

}

暴力生成法里数列

/\*\*

\* 数学上,n阶的法里数列是0和1之间最简分数的数列,由小至大

\* 排列,每个分数的分母不大于n.每个法里数列从0开始,至1结束,

\* 写作0/1和1/1.法里数列有许多奇妙的性质.以下程序能够输出

\* n阶法里数列.时间复杂度O(n^2),空间复杂度O(n)

\*/

void dfs(int x1, int y1, int x2, int y2) {

if (y1 + y2 <= n) {

dfs(x1, y1, x1 + x2, y1 + y2);

printf("%d/%d\t", x1 + x2, y1 + y2);

dfs(x1 + x2, y1 + y2, x2, y2);

}

}

void print\_farey(int n) {

printf("0/1\t");

dfs(0, 1, 1, 1);

printf("1/1\t");

}

求n阶法里数列第k项(k<n)

/\*\*

\* 为了能够在O(1)时间内求出n阶法里数列的第k项(k<n)，我对

\* 数列进行了仔细观察，发现如下规律：

\* ①这个数列中，最开始会有约n/2项是分子为1的(分母从n开

\* 始每项递减1，直到ceil(n/2.0)结束)；

\* ②接着会有约n/3项，一个分子为2的分数与一个分子为1分数

\* 交替出现，分子为2的数分母从n(若n%2!=0)或n-1(若n%2==0)

\* 开始每项递减2，直到floor(n/3.0 + 1/3.0) \* 2 + 1结束，

\* 分子为1的数的分母从上次结束的下一个位置

\* 即floor(n/2.0 + 1/2.0)-1开始每项递减1直到ceil(n/3.0)

\* 结束;

\* ③再然后还会有约n/3项，以3/a, x/b, 3/c, y/d的形式交替

\* 出现。其中a从n(或n-1，或n-2，即第一个不被3整除的数)开

\* 始每项递减3直到floor(n/4.0 + 1/4.0) \* 3 + 2结束。c从

\* a-1开始每项递减3直到直到floor(n/4.0 + 1/4.0) \* 3 + 1

\* 结束。剩下两项分两种情况，若a%3==2，则x=2，y=1，

\* 若a%3==1，则x=1，y=2。分子为2的项从上次结束的下一个位

\* 置即floor(n/3.0 + 1/3.0) \* 2 - 1开始每项递减2直到

\* floor(n/4.0+1/4.0) \* 2 + 1结束，分子为1的项从上次结束

\* 的下一个位置即floor(n/3.0+1/3.0)开始每项递减1直到

\* floor(n/4.0+1/4.0)结束。这些项加起来就有n/2+n/3+n/3>n

\* 项了，所以第k项一定在这些项里，按此规律即可出结果。

\*/

/\*\*

\* 得到n阶法里数列分子为1的连续段(到分子出现2时截止)长度

\*/

inline int getflen(int n) {

int t = (int)ceil(n / 2.0);

return n - t + 1;

}

/\*\*

\* 得到n阶法里数列分子为2和1交替出现的连续段(到分子出现3

\* 时截止)的长度

\*/

inline int getslen(int n) {

int s1 = n % 2 == 0 ? (n - 1) : n;

int t1 = (int)floor(n/3.0 + 1/3.0) \* 2 + 1;

int ans = (s1 - t1) / 2 + 1;

int s2 = (int)floor(n / 2.0 + 1 / 2.0) - 1;

int t2 = (int)ceil(n/3.0);

ans += s2 - t2 + 1;

return ans;

}

/\*\*

\* 得到n阶法里数列分子为3,1,2交替出现的连续段(到分子出现4

\* 时截止)的长度

\*/

inline int gettlen(int n) {

int s1 = n;

while(s1 % 3 == 0) {

s1--;

}

int t1 = (int)floor(n/4.0 + 1/4.0) \* 3 + 2;

int s3 = s1 - 1;

int t3 = (int)floor(n/4.0 + 1/4.0) \* 3 + 1;

int s2 = (int)floor(n/3.0 + 1/3.0) \* 2 - 1;

int t2 = (int)floor(n/4.0 + 1/4.0) \* 2 + 1;

int s4 = (int)ceil(n/3.0) - 1;

int t4 = (int)ceil(n/4.0);

int ans = 0;

if(s1 % 3 == 1) {

ans += 2;

s1 -= 2;

s3 -= 2;

s4 -= 1;

}

ans += (s1 - t1) / 3 + 1;

ans += (s3 - t3) / 3 + 1;

ans += (s2 - t2) / 2 + 1;

ans += s4 - t4 + 1;

return ans;

}

inline void get\_f(int n, int k, int &a, int &b) {

a = 1;

b = n - k + 1;

}

inline void get\_s(int n, int k, int &a, int &b) {

int s1 = n % 2 == 0 ? (n - 1) : n;

int s2 = (int)floor(n / 2.0 + 1 / 2.0);

if(k % 2 == 1) {

a = 2;

b = s1 - k + 1;

}else {

a = 1;

b = s2 - k / 2;

}

}

inline void get\_t(int n, int k, int &a, int &b) {

int s1 = n;

while(s1 % 3 == 0) {

s1--;

}

int s2 = (int)floor(n/3.0 + 1/3.0) \* 2 - 1;

int s3 = (s1 % 3 == 2) ? (s1 - 1) : (s1 - 2);

int s4 = (int)ceil(n/3.0) - 1;

if(k % 4 == 1) {

a = 3;

b = s1 - (k - 1) / 4 \* 3;

}else if(k % 4 == 3) {

a = 3;

b = s3 - (k - 3) / 4 \* 3;

}else if((k % 4 == 2) xor (s1 % 3 == 2)) {

a = 1;

if(k % 4 == 2) {

b = s4 - (k - 2) / 4;

}else {

b = s4 - (k - 4) / 4;

}

}else {

a = 2;

if(k % 4 == 2) {

b = s2 - (k - 2) / 4 \* 2;

}else {

b = s2 - (k - 4) / 4 \* 2;

}

}

}

/\*\*

\* 调用函数get\_farey(n, k)即得结果，结果的第一项为分子，

\* 第二项为分母。程序时间与空间复杂度均为O(1)

\*/

pair<int, int> get\_farey(int n, int k) {

pair<int, int> ret;

int fl = getflen(n);

int sl = getslen(n);

if(k <= fl) {

get\_f(n, k, ret.first, ret.second);

}else if(k <= sl + fl) {

get\_s(n, k - fl, ret.first, ret.second);

}else {

get\_t(n, k - fl - sl, ret.first, ret.second);

}

return ret;

}

巧求阶乘位数(斯特林公式)

求N的阶乘的位数

利用斯特林[stirling]公式(求阶乘(n!)的位数)

N\*log10(N/e)+log10(sqrt(2\*pi\*N))+1

另一种方式： log10(1)+log10(2)+````+log10(N)+1

#define e exp(1.0)

#define pi acos(-1.0)

int main() {

int T, N, bit;

while (scanf("%d", &T) != EOF) {

while (T--) {

scanf("%d", &N);

bit = (int) ((N \* (log(N) - log(e)) + 0.5 \* log(pi \* 2 \* N))

/ log(10));

printf("%d\n", bit + 1);

}

}

return 0;

}

大数取模

int getModBigNum(int p, char \*ch) {

int i, len;

LL res;

len = strlen(ch);

for (i = 0, res = 0; i < len; i++) {

res = (res \* 10 + (ch[i] - '0')) % p;

}

return (int) res;

}

快速幂a^b

LL getPow(int a, int b) {

LL res, temp;

res = 1, temp = (LL) a;

while (b) {

if (b & 1) {

res = res \* temp;

}

b >>= 1;

temp = temp \* temp;

}

return res;

}

乘积取余a\*b%c

防止a\*b超出了LL的数据类型

LL modular\_multi(LL a, LL b, LL c) {

LL res,temp;

res = 0,temp=a%c;

while (b) {

if (b & 1) {

res += temp;

if (res >= c) {

res -= c;

}

}

temp <<= 1;

if (temp >= c) {

temp -= c;

}

b >>= 1;

}

return res;

}

快速幂取余a^b%c

//这个版本只适用于a、b、c为int，long long可能溢出

int modular\_exp(int a, int b, int c) {

LL res, temp;

res = 1 % c, temp = a % c;

while (b) {

if (b & 1) {

res = res \* temp % c;

}

temp = temp \* temp % c;

b >>= 1;

}

return (int) res;

}

//以下为long long不会溢出的版本

typedef long long LL;

//计算a \* b % c

LL modular\_multi(LL a, LL b, LL c) {

LL res, t;

res = 0, t = a % c;

while (b) {

if (b & 1) {

res += t;

if (res >= c) {

res -= c;

}

}

t <<= 1;

if (t >= c) {

t -= c;

}

b >>= 1;

}

return res;

}

LL modular\_exp(LL a, LL b, LL c) {

LL res, t;

res = 1 % c, t = a % c;

while (b) {

if (b & 1) {

res = modular\_multi(res, t, c);

}

t = modular\_multi(t, t, c);

b >>= 1;

}

return res;

}

快速幂取余a^b%c（a,b很大）

int cmpBigNum(int p, char \*ch) {

int i, len;

LL res;

len = strlen(ch);

for (i = 0, res = 0; i < len; i++) {

res = (res \* 10 + (ch[i] - '0'));

if (res > p) {

return 1;

}

}

return 0;

}

void solve(int a, int c, char \*ch) {

int phi, res, b;

phi = getPhi(c);

if (cmpBigNum(phi, ch)) {

b = getModBigNum(phi, ch) + phi;

} else {

b = atoi(ch);

}

res = modular\_exp(a, b, c);

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

}

int main() {

int a, c;

char cha[nnum], chb[nnum];

mkprime();

while (~scanf("%s %s %d", cha, chb, &c)) {

a = getModBigNum(c, cha);

solve(a, c, chb);

}

return 0;

}

筛选素数

// 方法一

#define nmax 1000001

int flag[nmax], prime[nmax],plen;

void mkprime() {

int i, j;

memset(flag, -1, sizeof(flag));

for (i = 2, plen = 0; i < nmax; i++) {

if (flag[i]) {

prime[plen++] = i;

}

for (j = 0;(j < plen) &&(i \* prime[j] < nmax); j++) {

flag[i \* prime[j]] = 0;

if (i % prime[j] == 0) {

break;

}

}

}

}

// 方法二

#define nmax 1500

int prime[nmax],plen;

void mkprime() {

int i, j;

memset(prime, -1, sizeof(prime));

for (i = 2; i < nmax; i++) {

if (prime[i]) {

for (j = i + i; j < nmax; j += i) {

prime[j] = 0;

}

}

}

for (i = 2, plen = 0; i < nmax; i++) {

if (prime[i]) {

prime[plen++] = i;

}

}

}

得到区间素数

#define nmax 46341

#define dmax 1000005

int plen, res;

unsigned prime[nmax], mark[dmax];

void solve(unsigned a,unsigned b) {

if (a < 2) {

res = 2;

}

int i, j, d;

int te = (int) (sqrt(b \* 1.0));

d = b - a + 1;

memset(mark, 1, sizeof(mark));

for (i = 0; i < plen && prime[i] <= te; i++) {

if ((j = prime[i] \* (a / prime[i])) < a) {

j += prime[i];

}

if (j < prime[i] \* prime[i]) {

j = prime[i] \* prime[i];

}

for (; j <= b; j += prime[i]) {

mark[j - a] = 0;

}

}

for (i = 0; i < d; i++) {

if (mark[i]) {

res++;

}

}

}

米勒-罗宾素数测试

//直接调用miller\_rabin判断n是否是素数，也可以增加s的值以减少误判率，不过一般赛题情况下s=50已经足够

int witness(int a, int n) {

LL x, d = 1,i = (LL)(ceil(log(n - 1.0) / log(2.0)) - 1);

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

x = d;

d = (d \* d) % n;

if (d == 1 && x != 1 && x != n - 1)

return 1;

if (((n - 1) & (1 << i)) > 0)

d = (d \* a) % n;

}

return (d == 1 ? 0 : 1);

}

int miller\_rabin(int n, int s = 50) {

if (n == 2) return 1;

if ((n % 2) == 0) return 0;

int j; LL a;

for (j = 0; j < s; j++) {

//rand()随机产生[0, RAND\_MAX)内的整数RAND\_MAX=32768

//直接%n产生不了[RAND\_MAX, n)的数,使用LL防止乘法溢出

a = (LL)rand() \* (n - 2) / RAND\_MAX + 1;

if (witness(a, n))

return 0;

}

return 1;

}

大数素数分解

方法一：

void Factor(long long n) {

long long d = 2;

while (true) {

if (n % d == 0) {

Pollard(d);

Pollard(n / d);

return;

}

d++;

}

}

void Pollard(long long n) {

if (n <= 0)

printf("error\n");

if (n == 1)

return;

if (Miller\_rabin(n)) {

factor[cnt++] = n;

return;

}

long long i = 0, k = 2, x, y, d;

x = y = rand() % (n - 1) + 1;

while (i++ < 123456) {

x = (Mul\_Mod(x, x, n) + n - 1) % n;

d = Gcd((y - x + n) % n, n);

if (d != 1) {

Pollard(d);

Pollard(n / d);

return;

}

if (i == k) {

y = x;

k \*= 2;

}

}

Factor(n);

}

方法二：

LL pollard\_rho(LL n, int c) {

LL x, y, d, i, k;

x = rand() % (n - 1) + 1;

y = x;

i = 1LL, k = 2LL;

while (1) {

i++;

x = (modular\_multi(x, x, n) + c) % n;

d = gcd(y - x, n);

if ((1 < d) && (d < n)) {

return d;

}

if (x == y) {

return n;

}

if (i == k) {

k <<= 1;

y = x;

}

}

return -1;

}

void findFactor(LL n, int c) {

if (n == 1) {

return;

}

if (miller\_rabin(n, 6)) {

if (n < pmin) {

pmin = n;

}

return;

}

LL p = n;

while (p >= n) {

p = pollard\_rho(p, c--);

}

findFactor(p, c);

findFactor(n / p, c);

}

GCD & LCM Inverse

**//** POJ2429 SCU2106

void solve() {

qsort(factor, flen + 1, sizeof(factor[0]), cmp);

num[0] = factor[0];

nnum = 0;

int i;

for (i = 0; i < flen; i++) {

if (factor[i] == factor[i + 1]) {

num[nnum] \*= factor[i + 1];

} else {

num[++nnum] = factor[i + 1];

}

}

}

void dfs(int s, LL sn, LL n) {

if (s == nnum + 1) {

if (minx == -1 || (sn + n / sn < minx)) {

if (gcd(sn, n / sn) == 1) {

minx = sn + n / sn;

ans = sn;

}

}

return;

}

dfs(s + 1, sn \* num[s], n);

dfs(s + 1, sn, n);

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("t.txt", "r", stdin);

#endif

LL a, b, n, x, y;

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

if (a == b) {

printf("%lld %lld\n", a, b);

continue;

}

n = b / a;

if (miller\_rabin(n, 10)) {

printf("%lld %lld\n", a, b);

continue;

}

flen = -1;

findFactor(n, 207);

solve();

minx = -1LL;

dfs(0, 1LL, n);

x = ans;

y = n / ans;

if (x > y) {

n = x, x = y, y = n;

}

printf("%lld %lld\n", x \* a, y \* a);

}

return 0;

}

反素数zoj2562 timus 1748

/\*

这里面我们有一个prime【16】的数组，为什么只要这几个素数呢，因为这几个素数的乘积大于10^16,

而且就反素数的性质来说

比如2^t1\*3^t2\*5^t3\*...p1^x\*\*\*p2^y,假设p1是大于prime[]中所有的素数的，

因为这几个素数的乘积大于10^16,如果

我们添加p1在这个连乘积式子里面，那么必然有至少一个prime[i]不在这个连乘积式子里面，

但是对于因子的总数目而言

我们在乎的是幂的大小，而非素因子的大小，也就是说如果素因子越大，反而会使因子的数目偏小。

这里引进反素数知识：

反素数第一点：g(x)表示 x含有因子的数目，设 0<i<=x 则g(i)<=x;

反素数第二个特性：2^t1\*3^t2^5^t3\*7^t4..... 这里有 t1>=t2>=t3>=t4...

证明：如果ti<tj，其中i<j,由于pi小于pj,那么pi^tj\*pj^ti<pi^ti\*pj^tj,这样就出现了

因子数目相同，但x更小的情况，与反素数的定义矛盾。

\*/

#define LLU unsigned long long

#define pnum 17

int prime[pnum] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,

59 };

LLU n, nmin;

int nmax;

void dfs(int pstart, int limit, LLU now, int nnum) {

int i;

LLU p = prime[pstart];

double nnow;

if (now > n) {

return;

}

if (nnum > nmax) {

nmin = now;

nmax = nnum;

}

if (nnum == nmax && now < nmin) {

nmin = now;

}

for (i = 1; i <= limit; p \*= prime[pstart], i++) {

nnow = (double) now;

if (nnow \* p > n) {

break;

} else {

dfs(pstart + 1, i, now \* p, nnum \* (i + 1));

}

}

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("data.in", "r", stdin);

#endif

int t;

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

while (t--) {

scanf("%llu", &n);

nmin = 1, nmax = 1;

dfs(0, 60, 1, 1);

printf("%llu %d\n", nmin, nmax);

}

}

return 0;

}

求出N的所有素因子及其个数

/\*\*

\* p为素数表，存下sqrt(N)的素数即可，f保存结果

\* f[i].first表示N的一个素因数,f[i].second为这个素因子的个数

\* typec可以是int、long、long long等

\*/

typedef vector<pair<LL, typec> > Int\_Pair;

void get\_prime\_factor(typec N, Int\_Pair &f, const vector<int> &p) {

int i, t, n, pl = p.size();

f.clear();

for(i = 0; i < pl; i++) {

t = p[i];

if(N % t == 0) {

n = 0;

while(N % t == 0) {

n++; N /= t;

}

f.push\_back(make\_pair(t, n));

}

if(N == 1) { break; }

}

if(N > 1) {

f.push\_back(make\_pair(N, 1));

}

}

求N的所有约数

/\*\*

\* 求出N的所有约数，用法如下work函数所示。先打出素数表，

\* 求出N的所有素因子及个数，然后用搜索的方法求出所有的

\* 约数。参数中k表示当前处理到第几个素因子，now表示当前

\* 的约数，d是一个向量，临时存放每个素因子取多少个。ret

\* 存放最后的结果。typec可以是int、long、long long等

\*/

void dfs(int k, typec now, Int\_Pair &f, vector<int> &d, vector<typec> &ret) {

if(k == (int)f.size()) {

ret.push\_back(now);

return ;

}

d[k] = 0;

while(d[k] <= f[k].second) {

dfs(k + 1, now, f, d, ret);

d[k]++;

now \*= f[k].first;

}

}

int work(int N) {

vector<int> prime\_table;

Int\_Pair factor;

get\_prime\_table((int)sqrt(N) + 1, prime\_table);

get\_prime\_factor(N, factor, prime\_table);

vector<int> data;

data.resize(factor.size());

vector<int> result;

result.clear();

dfs(0, 1, factor, data, result);

return 0;

}

[求N的阶乘约数的个数](http://www.cppblog.com/jie414341055/articles/113194.html)

/\*先说一个定理： 若正整数n可分解为p1^a1\*p1^a2\*...\*pk^ak 其中pi为两两不同的素数,ai为对应指数 n的约数个数为(1+a1)\*(1+a2)\*....\*(1+ak)

如180=2\*2\*3\*3\*5=2^2\*3^2\*5

180的约数个数为1+2)\*(1+2)\*(1+1)=18个。

若求A/B的约数个数，A可分解为p1^a1\*p2^a2\*...\*pk^ak， B可分解为q1^b1\*q1^b2\*...\*qk^bk,则A/B的约数个数为(a1-b1+1)\*(a2-b2+1)\*(a3-b3+1)...\*(ak-bk+1).

然后说N的阶乘：

例如:20!

1.先求出20以内的素数,(2,3,5,7,11,13,17,19)

2.再求各个素数的阶数

e(2)=[20/2]+[20/4]+[20/8]+[20/16]=18;

e(3)=[20/3]+[20/9]=8;

e(5)=[20/5]=4;

...

e(19)=[20/19]=1;

所以 20!=2^18\*3^8\*5^4\*...\*19^1

解释： 2、4、6、8、10、12、14、16、18、20能被2整除 4、8、12、16、20能被4整除（即被2除一次后还能被2整除） 8、16能被8整除（即被2除两次后还能被2整除）

16能被16整除（即被2除三次后还能被2整除）

这样就得到了2的阶。其它可以依次递推。

所以在求N的阶乘质数因数个数时，从最小的质数开始，

递归：

int getNum(int n, int p) {

if(n < p) return 0;

else return n / p + getNum(n / p, p);

}

非递归：

int getNum( int n, int p) {

int res;

res = 0;

while (n) {

res += n / p;

n /= p;

}

return res;

}

其中P是质数，则该函数返回的就是N的阶乘中可以表达成质数P的指数的最大值。原理如上。

//获得 p在1\*……\*n中的个数

int getNum(int n, int p) {

int res;

res = 0;

while (n) {

res += n / p;

n /= p;

}

return res;

}

void solve(int n) {

int i, len\_factor;

for (i = 0, len\_factor = 0; (i < plen) && (prime[i] <= n); i++) {

pfactor[len\_factor] = prime[i];

cpfactor[len\_factor ++] = getNum(n, prime[i]);

}

printf("%d=", n);

printf("%d", pfactor[0]);

if (cpfactor[0] > 1) {

printf("^%d", cpfactor[0]);

}

for (i = 1; i < len\_factor; i++) {

printf("\*%d", pfactor[i]);

if (cpfactor[i] > 1) {

printf("^%d", cpfactor[i]);

}

}

printf("\n");

}

最大公约数(GCD)欧几里德算法

//typec可以为short ing、int、long、long long等

typec gcd(typec a, typec b) {

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

}

typec gcd(typec a, typec b) {

typec r;

while (b) {

r = a % b;

a = b, b = r;

}

return a;

}

“快速” GCD

//传统的GCD算法的时间复杂度是O(log(b))级的，但是其中涉及模除运算，故有学者提出如下的“快速”GCD算法，其特点主要是消除了传统算法中的模除运算，但个人觉得这种做法对算法的时间优化不是很明显，作为普通的程序设计竞赛，传统的欧几里德算法已经够用。放此代码仅供参考。

int gcd(int a, int b) {

if (a < b) { a ^= b, b ^= a, a ^= b; }

if (b == 0) { return a; }

if (!(a & 1) && !(b & 1))

return gcd(a >> 1, b >> 1) << 1;

else if (!(b & 1))

return gcd(a, b >> 1);

else if (!(a & 1))

return gcd(a >> 1, b);

else

return gcd(b, a - b);

}

扩展欧几里德

//求一组x, y使得gcd(a, b) = a \* x + b \* y;

//递归实现

int extend\_gcd(int a, int b, int &x, int &y) {

int d, xx;

if (b == 0) {

x = 1, y = 0;

return a;

}

d = extend\_gcd(b, a % b, x, y);

xx = x;

x = y, y = xx - a / b \* y;

return d;

}

//循环实现

int extend\_gcd(int a, int b, int &x, int &y) {

int g, v, w, t, q, s;

long long temp;

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

x = 1, g = a, v = 0, w = b;

while(w) {

q = g / w; t = g % w;

s = x - q \* v;

x = v, g = w, v = s, w = t;

}

temp = g;

temp -= ((long long)a) \* x;//防止溢出

y = temp / b;

return g;

}

模线性方程

//a \* x = b (mod n)

void modular\_equation(int a, int b, int n) {

int e, i, d;

d = extend\_gcd(a, n);

if (b % d) {

puts("No answer!");

return;

}

e = (x \* (b / d) % n + n) % n;

for (i = 0; i < d; i++) {

printf("The %dth answer is: %d\n", i + 1, (e + i \* (n / d)) % n);

}

printf("\n");

}

方程 ax = b (mod n) 有解, 当且仅当 gcd(a, n) | b;

方程 ax = b (mod n) 有d个不同的解, 其中 d = gcd(a, n);

方程有一解是: x0 = x'(b/d) mod n;

方程有d个解: xi = x0 + i\*(n/d) (mod n);

这样我们就可以求出方程的所有解了，但实际问题中，

我们往往被要求去求最小整数解，所以我们就可以将一个特解x，

t=b/（a，b），x=(x%t+t）%t；就可以了。

当x+b时，y-a才能保证原式成立。

b/=d,n/=d;

e = (x \* b % n + n) % n;

for (i = 0; i < d; i++) {

printf("The %dth answer is: %d\n", i + 1, (e + i \* n );

}

递推求欧拉函数值

#define maxn 1000

int phi[maxn];

void init() {

int i, j;

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

phi[i] = i;

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

phi[i] /= 2;

for (i = 3; i <= maxn; i += 2)

if (phi[i] == i) {

for (j = i; j <= maxn; j += i)

phi[j] = phi[j] / i \* (i - 1);

}

}

单独求欧拉函数值

/\*\*

\* 欧拉函数φ(n)表示[1, n]中与n互质的数的个数

\* typec可以是short int、int、long、long long等

\*/

typec getPhi(typec n) {

typec i, te, phi;

te = (typec) sqrt(n + 0.0);

for (i = 2, phi = n; i <= te; i++)

if (n % i == 0) {

phi = phi / i \* (i - 1);

while (n % i == 0) {

n /= i;

}

}

if (n > 1) {

phi = phi / n \* (n - 1);

}

return phi;

}

利用素数表求欧拉函数值

int getPhi(int n) {

int i, te, phi;

te = (int) sqrt(n \* 1.0);

for (i = 0, phi = n; (i < plen) && (prime[i] <= te); i++) {

if (n % prime[i] == 0) {

phi = phi / prime[i] \* (prime[i] - 1);

while (n % prime[i] == 0) {

n /= prime[i];

}

}

}

if (n > 1) {

phi = phi / n \* (n - 1);

}

return phi;

}

求欧拉函数的和

#define nmax 3000001

int prime[nmax], phi[nmax];

LL phiSum[nmax];

int plen;

void init() {

int i, j;

memset(phi, 0, sizeof(phi));

for (i = 2, plen = 0; i < nmax; i++) {

if (!phi[i]) {

phi[i] = i - 1;

prime[plen++] = i;

}

for (j = 0; (j < plen) && (i \* prime[j] < nmax); j++) {

if (i % prime[j] == 0) {

phi[i \* prime[j]] = phi[i] \* prime[j];

break;

} else {

phi[i \* prime[j]] = phi[i] \* (prime[j] - 1);

}

}

}

phiSum[1] = 0;

for (i = 2; i < nmax; i++) {

phiSum[i] = phiSum[i - 1] + phi[i];

}

}

满足gcd(n,i)=1的个数

//hdu 1695 其中0<i<m

题目意思不难已知给定k,x,y求 1<=a<=x 1<=b<=y 中满足 gcd(a,b)=k 的(a,b)对数。

（注意数对是无序的）。 1<=x,y<=10w, 0<=k<=10w

题目有比较恶心的一点，数据有k==0的，这时显然答案是0，没有2个数的gcd为0。

首先，gcd是没啥用的。因为约掉gcd后两个数互质。于是我们可以让x/=k y/=k并且假设 x<=y

然后题目变成了 2个数分别在区间[1..x]和[1..y]中的互质数有多少对。

大体思路：

枚举[1..y]中每个数i 判断[1..min(x,i)]中有多少数与i互质，统计个数。（注意，枚举的是比较大的区间[1..y]）。

显然如果i是质数，则[1..min(x,i)]中与i互质的个数是全体的个数或者i-1个。（取决于x和i的大小）。

当i不是质数时，i分解质因数后，质因数的次数不影响结果。我们看另外那个区间有多少个和i不互质（减一下就好了），于是我们只要看另外那个区间中有多少个数是i质因数的倍数就好了。

区间[1..w]中 p的倍数 显然有 w/p个。

我们枚举i的质因数利用容斥原理：

看另外那个区间有多少个数与i不互质。

容斥原理的具体如下：

区间中与i不互质的个数 = （区间中i的每个质因数的倍数个数）-（区间中i的每两个质因数乘积的倍数）+（区间中i的每3个质因数的成绩的倍数个数）-（区间中i的每4个质因数的乘积）+...

于是问题变成了统计每个数的不同质因数的个数而忽略次数。这个可以用筛法。具体做法如下：

对每个数保存一个真质因数的列表。初始每个列表的长度为0。然后从2开始，分别检查每个数的列表长度，如果列表长度不为0，则这个数是合数，跳过；如果这个长度为0，则我们找到了一个质数，同时再把这个数的倍数（不包含本身）的列表里加入这个数。

这样筛一次下来，我们保存了每个数的真质因数列表，问题得到解决，还要注意结果用要用

LL dfs(int n, int start) {

int i;

LL res;

for (i = start, res = 0; i < pflen; i++) {

res += n / pfactor[i] - dfs(n / pfactor[i], i + 1);

}

return res;

}

void solve(int b, int d, int k) {

int i;

LL res;

if (k == 0 || k > b || k > d) {

puts("0");

return;

}

b /= k, d /= k;

if (b > d) {

b ^= d, d ^= b, b ^= d;

}

for (i = 1, res = 0; i <= b; i++) {

res += phi[i];

}

for (i = b + 1; i <= d; i++) {

getpFactor(i);

res += b - dfs(b, 0);

}

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

}

满足a^x=1(mod n)最小的x

//其中 (a,n)=1

//方法一：分解n的欧拉函数值，取最小的满足原式，就是答案

void getRemainOne(int a, int n) {

int i, te, phi, res;

phi = getPhi(n);

te = (int) sqrt(phi \* 1.0);

for (i = 1, res = phi; i <= te; i++) {

if (phi % i == 0) {

if ((modular\_exp(a, i, n) == 1) && (i < res)) {

res = i;

}

if ((modular\_exp(a, phi / i, n) == 1) && (phi / i < res)) {

res = phi / i;

}

}

}

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

}

//方法二：见下面的例子。

poj 3696 M=mmmm M%k==0

求解10^x = 1 (mod 9\*L/gcd(L,8))的满足x>0的最小解就是答案

由8构成的数A设有x位

那么A=8(10^0+10^1+...+10^(x-1));

很容易得到A=(8/9)\*(10^x-1);

题目的要求就是A=0(mod L)

就是(8/9)\*(10^x-1)=0(mod L);

->8\*(10^x-1)=0(mod 9L);

->10^x-1=0(mod 9L/gcd(L,8));

->10^x =1 (mod 9L/gcd(L,8));

令p=9\*L/gcd(L,i), 等价于10^x==1(mod p),求满足条件的最小的整数x

看到这个式子，我们的第一反应就是数论书上的Euler定理，但是如果gcd（10，p)!=1,即不互素，

也就是不满足欧拉定理的条件了，所以此时为no answer.

if(gcd（10，p)==1) 满足欧拉定理, 我们知道10^oula(p)==1(mod p)肯定是满足的;

虽然oula(p)满足上述同余方程，但题目求是最小x，而oula(p)未必是最小的解。考虑到oula(p)不是素数，

所以我们就要将oula(p)因子分解，求出oula(p)所有的因子，然后逐个从小到大枚举，如果满足10^x==1(mod p) (其中x为oula(p)的因子)，x即为答案。

#define nmax 134165

#define nnum 13000

#define num8 8

#define num9 9

#define num10 10

int flag[nmax], prime[nnum], cpfactor[nnum];

LL pfactor[nnum], factor[nnum];

int plen, len\_pfactor, len\_factor;

void solve(int n) {

int i, d;

LL c, phi;

// if (n == 1) {

// puts("1");

// return;

// }

// if (n % 16 == 0 || n % 5 == 0) {

// puts("0");

// return;

// }

d = gcd(n, num8);

c = (LL) n;

c = c / d \* num9;

if (gcd(num10, c) != 1) {

puts("0");

return;

}

phi = getPhi(c);

findpFactor(phi);

len\_factor = 0;

dfs(0, 1);

qsort(factor, len\_factor, sizeof(factor[0]), cmp);

for (i = 0; i < len\_factor; i++) {

if (modular\_exp(num10, factor[i], c) == 1) {

printf("%I64d\n", factor[i]);

return;

}

}

}

最小公倍数和

g(n)=∑{phi(n/d)\*n/d},d|n

f(n)=(g(n)+1)/2

由于欧拉函数的积性，phi(n\*m)=phi(n)\*phi(m),gcd(n,m)=1

所以，n=∏(pi^ci)

故g(n)=g(∏(pi^ci))。

g(n)= ∏(g(pi^ci)

g(pi^ci)= ∑(phi(pi^ci)pi^ci)

而对于phi(pi^ci)=(pi-1)\*(pi^(ci-1)

方法一：

LL sumlcm(int n) {

int te = 0, temp = n;

LL sum = 0;

if (n == 1)

return 1;

te = (int) sqrt(n \* 1.0);

sum = (LL) n;

for (int i = 2; i <= te; i++) {

if (n % i == 0) {

sum = sum / i \* (i - 1);

while (n % i == 0) {

n /= i;

}

}

}

if (n > 1) {

sum = sum / n \* (n - 1);

}

return sum \* temp / 2;

}

void solve(int n) {

int i, te, temp;

LL sum;

te = (int) sqrt(n \* 1.0);

for (i = 1, sum = 0; i <= te; i++) {

if (n % i == 0) {

temp = n / i;

sum += sumlcm(temp);

if (i != temp) {

sum += sumlcm(i);

}

}

}

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

}

方法二：

A：

LL calc(int p, int c) {

LL res;

res = getPow(p, 2 \* c);

res--;

res = res / (p + 1);

res = res \* p;

res++;

return res;

}

B.

int getpPhi(int p, int c) {

if (c == 0) {

return 1;

}

return (p - 1) \* getPow(p, c - 1);

}

LL calc(int p, int c) {

int i;

LL temp, sum;

for (i = 0, sum = 0LL, temp = 1LL; i <= c; i++) {

sum += temp \* getpPhi(p, i);

temp = temp \* p;

}

return sum;

}

//核心处理部分

void solve(int n) {

int i, te, cnt;

LL res;

te = (int) (sqrt(n \* 1.0));

for (i = 0, res = 1LL; (i < plen) && (prime[i] <= te); i++) {

if (n % prime[i] == 0) {

cnt = 0;

while (n % prime[i] == 0) {

cnt++;

n /= prime[i];

}

res = res \* calc(prime[i], cnt);

}

}

if (n > 1) {

res = res \* calc(n, 1);

}

printf("%lld\n", (res + 1) / 2);

}

模线性方程组

//a=B[1](% W[1]); a=B[2](% W[2]); ... a=B[k](% W[k]);

//其中W, B已知，W[i]>0且W[i]与W[j]互质, 求a; (中国余数定理)

int china(int b[], int w[], int k) {

int i, d, x, y, m, a = 0, n = 1;

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

n \*= w[i]; // ! 注意不能overflow

for (i = 0; i < k; i++) {

m = n / w[i];

d = extend\_gcd(w[i], m, x, y); //扩展欧几里德

a = (a + y \* m \* b[i]) % n;

}

if (a > 0)

return a;

else

return (a + n);

}

离散对数（discrete logging）

特殊Baby Step Giant Step

pku 2417

【问题模型】 求解 A^x = B (mod C) 中 0 <= x < C 的解,C 为素数

【思路】 我们可以做一个等价

x = i \* m + j ( 0 <= i < m, 0 <=j < m) m = Ceil ( sqrt( C) )

而这么分解的目的无非是为了转化为:

(A^i)^m \* A^j = B ( mod C)

之后做少许暴力的工作就可以解决问题：

(1) for i = 0 -> m, 插入Hash (i, A^i mod C)

(2) 枚举 i ,对于每一个枚举到的i,令 AA = (A^m)^i mod C

我们有 AA \* A^j = B (mod C)

显然AA,B,C均已知，而由于C为素数,那么(AA,C)无条件为1

于是对于这个模方程解的个数唯一(可以利用扩展欧几里得或 欧拉定理来求解)

那么对于得到的唯一解X,在Hash表中寻找,如果找到，则返回 i \* m + j

注意:由于i从小到大的枚举,而Hash表中存在的j必然是对于某个剩余系内的元素X 是最小的(就是指标)

所以显然此时就可以得到最小解

如果需要得到 x > 0的解，那么只需要在上面的步骤中判断 当 i \* m + j > 0 的时候才返回

到目前为止，以上的算法都不存在争议,大家实现的代码均相差不大。可见当C为素数的时候，

此类离散对数的问题可以变得十分容易实现。

#define nmax 46345

typedef struct num {

int ii, value;

} num;

num Num[nmax];

int bfindNum(int key, int n) {

int left, right, mid;

left = 0, right = n;

while (left <= right) {

mid = (left + right) >> 1;

if (Num[mid].value == key) {

return Num[mid].ii;

} else if (Num[mid].value > key) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return -1;

}

void solve(int c, int a, int b) {

int i, j, te, aa;

LL temp, xx;

te = (int) (sqrt(c \* 1.0) + 0.5);

for (i = 0, temp = 1 % c; i <= te; i++) {

Num[i].ii = i;

Num[i].value = (int) (temp);

temp = temp \* a % c;

}

aa = Num[te].value;

qsort(Num, te + 1, sizeof(Num[0]), cmp);

for (i = 0, temp = 1; i <= te; i++) {

extend\_gcd((int) (temp), c);

xx = (LL) x;

xx = xx \* b;

xx = xx % c + c;

x = (int) (xx % c);

j = bfindNum(x, te + 1);

if (j != -1) {

printf("%d\n", i \* te + j);

return;

}

temp = temp \* aa % c;

}

puts("no solution");

}

一般Baby Step Giant Step

【问题模型】 求解 A^x = B (mod C) 中 0 <= x < C 的解,C 无限制(当然大小有限制……)

【写在前面】 这个问题比较麻烦，目前网络上流传许多版本的做法，不过大部分已近被证明是完全错误的！

这里就不再累述这些做法，下面是我的做法(有问题欢迎提出) 下面先给出算法框架，稍后给出详细证明:

(0) for i = 0 -> 50 if(A^i mod C == B) return i O(50)

(1) cd<- 0 temp<- 1 mod C

while((d=gcd(A,C))!=1)

{

if(B%d)return -1; // 无解!

++cd;

C/=d;

B/=d;

temp=temp\*A/d%C;

}

(2) te = Ceil ( sqrt(C) ) //Ceil是必要的 O(1)

(3) for i = 0 -> te 插入Hash表(i, A^i mod C) O(te)

(4) aa=pow\_mod(A,te,C)

for i = 0 -> te

解 temp \* X = B (mod C) 的唯一解 (如果存在解，必然唯一!)

之后Hash表中查询,若查到(假设是 j)，则 return i \* te + j + cd

否则

temp=temp\*aa%C,继续循环

(5) 无条件返回 -1 ;//无解!

下面是证明:

推论1:

A^x = B(mod C)

等价为

A^a \* A^b = B ( mod C) (a+b) == x a,b >= 0

证明:

A^x = temp \* C + B (模的定义)

A^a \* A^b = temp\*C + B( a,b >=0, a + b == x)

所以有

A^a \* A^b = B(mod C)

推论 2:

令 AA \* A^b = B(mod C)

那么解存在的必要条件为: 可以得到至少一个可行解 A^b = X (mod C)

使上式成立

推论3

AA \* A^b = B(mod C)

中解的个数为 (AA,C)

由推论3 不难想到对原始Baby Step Giant Step的改进

For I = 0 -> m

For any solution that AA \* X = B (mod C)

If find X

Return I \* m + j

而根据推论3,以上算法的复杂度实际在 (AA,C)很大的时候会退化到几乎O(C)

归结原因，是因为(AA,C)过大，而就是(A,C)过大

于是我们需要找到一中做法，可以将(A,C)减少，并不影响解

下面介绍一种“消因子”的做法

一开始temp = 1 mod C

进行若干论的消因子,对于每次消因子

令 d = (A,C[i]) // C[i]表示经过i轮消因子以后的C的值

如果不存在 d | B[i] //B[i]表示经过i轮消因子以后的B的值

直接返回无解

否则

B[i+1] = B[i] / d

C[i+1] = C[i] / d

temp = temp \* A / d

具体实现只需要用若干变量,细节参考代码

假设我们消了a'轮(假设最后得到的B,C分别为B',C')

那么有

temp \* A^b = B' (mod C')

于是可以得到算法

for i = 0 -> m

解 ( temp\* (A^m) ^i ) \* X = B'(mod C')

由于 ( temp\* (A^m) ^i , C') = 1 (想想为什么？)

于是我们可以得到唯一解

之后的做法就是对于这个唯一解在Hash中查找

这样我们可以得到b的值,那么最小解就是a' + b !!

现在问题大约已近解决了，可是细心看来，其实还是有BUG的，那就是

对于

A^x = B(mod C)

如果x的最小解< a',那么会出错

而考虑到每次消因子最小消 2

故a'最大值为log(C)

于是我们可以暴力枚举0->log(C)的解，若得到了一个解，直接返回

否则必然有 解x > log(C)

PS.以上算法基于Hash 表,如果使用map等平衡树维护，那么复杂度会更大

pku 3243

#define nnum 100

#define nmax 31625

typedef struct num {

int ii, value;

} num;

num Num[nmax];

/\* ax=b (mod n)\*/

int inval(int a, int b, int n) {

LL res;

extend\_gcd(a, n);

res = (LL) x;

res = res \* b;

res = (res % n + n) % n;

return (int) res;

}

int bfindNum(int key, int n) {

int left, right, mid;

left = 0, right = n;

while (left <= right) {

mid = (left + right) >> 1;

if (Num[mid].value == key) {

return Num[mid].ii;

} else if (Num[mid].value > key) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return -1;

}

void baby\_step\_giant\_step(int a, int b, int c) {

int i, j, te, d, cd, aa, ttemp;

LL temp, tem;

for (i = 0, temp = 1 % c, tem = temp; i < nnum; i++, temp = temp \* a % c) {

if (temp == b) {

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

return;

}

}

cd = 0;

while ((d = gcd(a, c)) != 1) {

if (b % d) {

puts("No Solution");

return;

}

cd++;

c /= d, b /= d;

tem = tem \* a / d % c;

}

te = (int) (sqrt(c \* 1.0) + 0.5);

for (i = 0, temp = 1 % c; i <= te; i++, temp = temp \* a % c) {

Num[i].ii = i;

Num[i].value = (int) temp;

}

aa = Num[te].value;

qsort(Num, te + 1, sizeof(Num[0]), cmp);

for (i = 0; i <= te; i++, tem = tem \* aa % c) {

ttemp = inval(tem, b, c);

if (ttemp >= 0) {

j = bfindNum(ttemp, te + 1);

if (j != -1) {

printf("%d\n", i \* te + j + cd);

return;

}

}

}

puts("No Solution");

}

x^2=n (mod m) 是不是二次剩余

考虑形如x2≡n（mod m）的同余式，其中m > 1，（m，n）＝1。

若此同余式有解，则n称为模m的二次剩余；若此同余式无解，则n称为模m的二次非剩余。

设p是一个奇素数，则模p的二次剩余和二次非剩余个数正好是“一半对一半”，

下表给出几个较小的素数模的二次剩余和非剩余：

> p 剩余 非剩余 > 3 1 2 > 5 1，4 2，3 > 7

1，2，4 3，5，6 > 11 1，3，4，5，9 2，6，7，8，10 > 13

\*> 1，3，4，9，11，12 2，5，6，7，8，11 > 此外，

如果n是模p的二次剩余，则N^((p-1)/2)≡1(mod p) 。如果n是模p的二次非剩余，

则N^((p-1)/2)≡-1(mod p) 。

void solve(int a, int p) {

a = (a % p + p) % p;

if (modular\_exp(a, (p - 1) / 2, p) == 1) {

puts("1");

return;

}

puts("-1");

}

X^k = n mod p (p is prime)

模素数p的原根g的优美体现在每个模p的非零数以g的幂次出现。所以，对任何数1 <= a < p，我们可选择幂

g,g^2,g^2,```````,g^(p-2),g^(p-1)

中恰好一个与a模p同余。相应的指数被称为以g为底的a模p的指标。假设p与g已给定，则记指标为I(a)。

以下以2模13的所有幂的形式：

I　　　　　　　 1　 2　　3　 4　 5　　 6　　 7　　 8　 9　　10　 11　12

2^I(mod 13)　 2　　4　　8　　3　　6　　12　　11　　9　　5　　10　　7　　1

例如，为求I(11)，我们搜寻表的第二行直到找到数11，则指标I(11)=7可从第一行得到。

指标法则

a).　　I(ab)=I(a)+I(b) (mod p-1)

b).　　I(a^k)=kI(a) (mod p-1)

g^I(ab)=ab=g^I(a)g^I(b)=g^(I(a)+I(b)) (mod p)

故有g^I(x) = x (mod p-1)

也可以这么理解：g^I(value) = id (mod p-1)

例题：3\*x^30=4(mod 37)

I(3\*x^30)=I(4)

I(3)+30\*I(x)=I(4) (mod 36)

26+30\*I(x)=2 (mod 36)

30\*I(x)=-24=12 (mod 36)

对于这里I(4)=2解释一下：

其实就是g^2 = 4 (mod 36)

其实这个可以根据g^x =b (mod p)来求

对于本题，g就是37的原根2,b=I(x) p =36

也就是求满足2^x = I(x) (mod 36)的x。用Baby Step Giant Step算法即可哦

提醒：两边不要除以6以得到5\*I(x)+2 (mod 36),否则会丢失一些解。

ax = c (mod m)

由扩展欧几里德，知：

I(x)=4,10,16,22,28,34

最后，有指标表（书论书上有哦，自己不放写程序看看），得到x的对应值

I(16)=4,I(25)=10,I(9)=16

I(21)=22,I(12)=28,I(28)=34

其实这里，也可以根据指标的原式公式g^I(x) = x (mod p-1)

故，同余式3\*x^30=4 (mod 37)有6个解，即

x=16,25,9,21,12,28 (mod 37)

#define nmax 4001

typedef struct num {

int ii, value;

} num;

num Num[nmax];

int k, n, p, proot;

int dfs(int depth, LL now) {

int i;

LL res, temp;

if (depth == len\_pfactor) {

res = modular\_exp(proot, now, p);

if ((res == 1) && (now != (p - 1))) {

return 0;

}

return 1;

}

for (i = 0, temp = 1; i <= cpfactor[depth]; i++) {

if (!dfs(depth + 1, now \* temp)) {

return 0;

}

temp = temp \* pfactor[depth];

}

return 1;

}

void primitive() {

findpFactor(p - 1);

for (proot = 2;; proot++) {

if (dfs(0, 1)) {

return;

}

}

}

/\* ax = b (mod c)\*/

int result[nmax];

void solve(int a, int b, int c) {

int i, d, cc;

d = extend\_gcd(a, c);

if (b % d) {

puts("No Solution!");

return;

}

cc = c;

b /= d, c /= d;

result[0] = ((LL) x \* b % c + c) % c;

for (i = 1; i < d; i++) {

result[i] = (result[i - 1] + c) % cc;

}

for (i = 0; i < d; i++) {

result[i] = modular\_exp(proot, result[i], p);

}

qsort(result, d, sizeof(result[0]), rcmp);

for (i = 0; i < d; i++) {

printf("%d\n", result[i]);

}

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("data.in", "r", stdin);

#endif

int a, b, c;

mkprime();

/\*x^k = n (mod p) \*/

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

primitive();

b = baby\_step\_giant\_step(proot, n, p);

a = k, c = p - 1;

solve(a, b, c);

printf("\n");

}

return 0;

}

x^2=1 (mod n)解的个数

给出整数n，统计二次同余方程x^2=1(mod n)在[0,n)闭区间上的解的个数。

分析：

根据《简明数论》上的说明：

若p是素数，x^2=1(mod p^k)的解数有如下结论。

当p==2时，k==1则解数是1，k==2时解数是2，k>=3时，解数是4。

当p>2时，k>0时解数是2。

然后分解因数。

void solve(int n) {

int i, res;

if (n == 1) {

puts("0");

return;

}

findpFactor(n);

for (i = 0, res = 1; i < len\_factor; i++) {

if (pfactor[i] == 2) {

if (cpfactor[i] >= 3) {

res = res \* 4;

} else {

res = res \* cpfactor[i];

}

} else {

res = res \* 2;

}

}

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

}

小知识

能被3整除数的特征：各位数字和能被3整除

能被11整除数的特征：所有奇数位数字之和减去偶数位数字之和的值能被11整除

幻方构造

//幻方构造(l!=2)

#define MAXN 100

void dllb(int l, int si, int sj, int sn, int d[][MAXN]) {

int n, i = 0, j = l / 2;

for (n = 1; n <= l \* l; n++) {

d[i + si][j + sj] = n + sn;

if (n % l) {

i = (i) ? (i - 1) : (l - 1);

j = (j == l - 1) ? 0 : (j + 1);

} else

i = (i == l - 1) ? 0 : (i + 1);

}

}

void magic\_odd(int l, int d[][MAXN]) {

dllb(l, 0, 0, 0, d);

}

void magic\_4k(int l, int d[][MAXN]) {

int i, j;

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

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

d[i][j] =

((i % 4 == 0 || i % 4 == 3) && (j % 4 == 0 || j % 4 == 3)

|| (i % 4 == 1 || i % 4 == 2)

&& (j % 4 == 1 || j % 4 == 2)) ?

(l \* l - (i \* l + j)) : (i \* l + j + 1);

}

void magic\_other(int l, int d[][MAXN]) {

int i, j, t;

dllb(l / 2, 0, 0, 0, d);

dllb(l / 2, l / 2, l / 2, l \* l / 4, d);

dllb(l / 2, 0, l / 2, l \* l / 2, d);

dllb(l / 2, l / 2, 0, l \* l / 4 \* 3, d);

for (i = 0; i < l / 2; i++)

for (j = 0; j < l / 4; j++)

if (i != l / 4 || j)

t = d[i][j], d[i][j] = d[i + l / 2][j], d[i + l / 2][j] = t;

t = d[l / 4][l / 4], d[l / 4][l / 4] = d[l / 4 + l / 2][l / 4], d[l / 4

+ l / 2][l / 4] = t;

for (i = 0; i < l / 2; i++)

for (j = l - l / 4 + 1; j < l; j++)

t = d[i][j], d[i][j] = d[i + l / 2][j], d[i + l / 2][j] = t;

}

void generate(int l, int d[][MAXN]) {

if (l % 2)

magic\_odd(l, d);

else if (l % 4 == 0)

magic\_4k(l, d);

else

magic\_other(l, d);

}

1. 组合数学

打组合数表

//如下定义c数组，初始化后c[i][j]表示

const int MAXN = 105;

const int MAXV =10005;

int c[MAXV][MAXN];

void InitCombineNum(){

memset(c, 0, sizeof(c));

for (int i = 0; i < MAXV; i++) { c[i][0] = 1; }

c[1][1] = 1;

for (int i = 2; i < MAXV; i++) {

for (int j = 1; j < MAXN; j++) {

if(j == i) { c[i][j] = 1; break;}

c[i][j] = (c[i - 1][j - 1] + c[i - 1][j]) % MOD;

}

}

}

排列数的末尾非零数

int Table[4][4] = { { 6, 2, 4, 8 }, { 1, 3, 9, 7 }, { 1, 7, 9, 3 },

{ 1, 9, 1, 9 } };

int GetBase25(int a, int b) {

if (a == 0)

return 0;

return a / b + GetBase25(a / b, b);

}

int GetOdd(int n, int m) {

if (n == 0)

return 0;

return n / 10 + (n % 10 >= m) + GetOdd(n / 5, m);

}

int Get(int n, int m) {

if (n == 0)

return 0;

return Get(n / 2, m) + GetOdd(n, m);

}

int main() {

int n, m, num2, num5, num3, num7, num9, res;

while (scanf("%d %d", &n, &m) != EOF) {

num2 = GetBase25(n, 2) - GetBase25(n - m, 2);

num5 = GetBase25(n, 5) - GetBase25(n - m, 5);

num3 = Get(n, 3) - Get(n - m, 3);

num7 = Get(n, 7) - Get(n - m, 7);

num9 = Get(n, 9) - Get(n - m, 9);

res = 1;

if (num5 > num2) {

printf("5\n");

continue;

}

if (num2 != num5) {

res \*= Table[0][(num2 - num5) % 4];

res %= 10;

}

res \*= Table[1][(num3 % 4 + 4) % 4];

res %= 10;

res \*= Table[2][(num7 % 4 + 4) % 4];

res %= 10;

res \*= Table[3][(num9 % 4 + 4) % 4];

res %= 10;

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

}

return 0;

}

排列数组合数的末尾非零数

int GetBase25(int n, int m) {

if (n == 0)

return 0;

return n / m + GetBase25(n / m, m);

}

int GetBase(int n, int m) {

if (n == 0)

return 0;

return n / 10 + (n % 10 >= m) + GetBase(n / 5, m);

}

int Get(int n, int m) {

if (n == 0)

return 0;

return Get(n / 2, m) + GetBase(n, m);

}

int Table[4][4] = { { 6, 2, 4, 8 }, { 1, 3, 9, 7 }, { 1, 7, 9, 3 },

{ 1, 9, 1, 9 } };

int main() {

int n, m, d, num2, num5, num3, num7, num9, res;

while (scanf("%d %d", &n, &m) != EOF) {

d = n - m;

num2 = GetBase25(n, 2) - GetBase25(m, 2) - GetBase25(d, 2);

num5 = GetBase25(n, 5) - GetBase25(m, 5) - GetBase25(d, 5);

num3 = Get(n, 3) - Get(m, 3) - Get(d, 3);

num7 = Get(n, 7) - Get(m, 7) - Get(d, 7);

num9 = Get(n, 9) - Get(m, 9) - Get(d, 9);

if (num5 > num2) {

printf("5\n");

continue;

}

res = 1;

if (num5 != num2) {

res \*= Table[0][((num2 - num5) % 4 + 4) % 4];

res %= 10;

}

res \*= Table[1][(num3 % 4 + 4) % 4];

res %= 10;

res \*= Table[2][(num7 % 4 + 4) % 4];

res %= 10;

res \*= Table[3][(num9 % 4 + 4) % 4];

res %= 10;

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

}

return 0;

}

组合数取模C(n,m)%p

1） 当p是素数时，

A).直接暴力

C(n,m) = [n\*(n-1)\*(n-2)\*...\*(n-m+1)]/[1\*2\*3\*...\*m].

令 a = [n\*(n-1)\*(n-2)\*...\*(n-m+1)]; b = [1\*2\*3\*...\*m]; x = C(n,m)%p;

如果（a,p）=1,(b,p) = 1, 则：(a/b)≡x(mod p);

==>a≡b\*x(mod p); 令 a` = a%p; b` = b%p;

则：a`≡b`\*x(mod p) ==>b`\*x+k\*p = a` (p为素数，gcd(b`,p)=1,方程一定有解)

利用扩展欧几里德或欧拉函数即可。

注意：在求解过程中，要确保（a,p）=1,(b,p) = 1,也就是说要将a,b包含的素数p全部除去，并分别统计C(n,m)中分子分母包含p的个数。若分子中的p个数大于分母，直接输出0，相等则应用拓展欧几里得。不会出现分子中p的个数小于分母中p的个数的情况。

代码如下：

int getMultMod(int start, int end, int p) {

int i, j;

LL res;

for (i = start, res = 1, pnum = 0; i <= end; i++) {

if (i % p == 0) {

j = i;

while (j % p == 0) {

pnum++;

j /= p;

}

res = res \* j % p;

} else {

res = res \* i % p;

}

}

return (int) (res % p);

}

void C(int n, int m, int p) {

int a, b, apnum, bpnum;

LL res;

a = getMultMod(n - m + 1, n, p);

apnum = pnum;

b = getMultMod(1, m, p);

bpnum = pnum;

if (apnum > bpnum) {

puts("0");

return;

} else {

//方法一（扩展欧几里德）：

extend\_gcd(b, p);

res = (LL) x;

res = (res % p + p) % p;

//方法二（欧拉函数）：

//res = modular\_exp(b, p - 2, p);

res = res \* a % p;

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

}

}

B).利用Lucas定理

Lucas theorem

m = mk \* p^k + mk-1 \* p^k-1 +... +m1 \* p + m0;

n = nk \* p^k + nk-1 \* p^k-1 +... + n1 \* p + n0; C(m,n)=C(mk,nk)\*C(mk-1,nk-1)\*...\*C(m1,n1)\*C(m0,n0);

int C(int a, int b, int p) {

int i;

LL resa, resb, res;

if (b > a) {

return 0;

}

//以下是求小组合数

//方法一：直接算

for (i = 0, resa = 1, resb = 1; i < b; i++) {

resa = resa \* (a - i) % p, resb = resb \* (b - i) % p;

}

//方法二：打阶乘对P的余数表，将组合数化成阶乘的形式

//b = num[b] \* num[a - b] % nmod;

//a = num[a];

//以下是求逆元

//第一种方法

extend\_gcd(resb, p);

res = (LL) x;

res = (res % p + p) % p;

//第二种方法

//res = modular\_exp(resb, p - 2, p);

res = res \* resa % p;

return (int) res;

}

void solve(int n, int m, int p) {

int a, b;

LL res;

res = 1;

while (n || m) {

a = n % p, b = m % p;

res = res \* C(a, b, p) % p;

n /= p, m /= p;

}

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

}

当P是整数时，

void solve(int a, int b, int c) {

int i, temp;

LL res;

for (i = 0, res = 1; (i < plen) && (prime[i] <= a); i++) {

temp = getNum(a, prime[i]) - getNum(b, prime[i])

- getNum(a - b, prime[i]);

res = res \* modular\_exp(prime[i], temp, c) % c;

}

printf("%lld\n", res);

}

组合数C(n,m)与因数k

包含因数的个数

//将K化成素因子的乘积的形式

void findpFactor(int k) {

int i, te, cnt;

te = (int) sqrt(k \* 1.0);

for (i = 0, cplen = 0; (i < plen) && (prime[i] <= te); i++) {

if (k % prime[i] == 0) {

cnt = 0;

while (k % prime[i] == 0) {

cnt++;

k /= prime[i];

}

pfactor[cplen] = prime[i];

cpfactor[cplen++] = cnt;

} }

if (k > 1) {

pfactor[cplen] = k;

cpfactor[cplen++] = 1;

}

}

//获得素因子是在组合数中的个数，

//并凭借该素因子在因数k中的个数，得到该因数k在组合数中的个数

int cal(int i, int n, int m) {

int temp;

temp = getNum(n, pfactor[i]);

temp -= getNum(m, pfactor[i]);

temp -= getNum(n - m, pfactor[i]);

return temp / cpfactor[i];

}

void solve(int n, int m, int k) {

int i, temp, res;

//找k的素因子以及相应的个数

findpFactor(k);

for (i = 0, res = nmax; i < cplen; i++) {

temp = cal(i, n, m);

if (temp < res) {

res = temp;

}

}

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

}

Catalan Numbers(卡特兰数)

令h(1)＝1,h(0)=1，catalan数满足递归式： h(n)= h(0)\*h(n-1)+h(1)\*h(n-2) + ... + h(n-1)h(0) (其中n>=2)

另类递归式：h(n)=((4\*n-2)/(n+1))\*h(n-1);

该递推关系的解为： h(n)=C(2n,n)/(n+1) (n=1,2,3,...)

组合数学中有非常多的组合结构可以用卡塔兰数来计数。 在Richard P. Stanley的Enumerative Combinatorics: Volume 2一书的习题中包括了66个相异的可由卡塔兰数表达的组合结构。以下用Cn=3和Cn=4举若干例：

Cn表示长度2n的dyck word的个数。Dyck word是一个有n个X和n个Y组成的字串，且所有的部分字串皆满足X的个数大于等于Y的个数。以下为长度为6的dyck words:

XXXYYY XYXXYY XYXYXY XXYYXY XXYXYY将上例的X换成左括号，Y换成右括号， Cn表示所有包含n组括号的合法运算式的个数: ((())) ()(()) ()()() (())() (()())

Cn表示有n+1个叶子的二叉树的个数。

Cn表示所有不同构的含n个分枝结点的满二叉树的个数。

（一个有根二叉树是满的当且仅当每个结点都有两个子树或没有子树。）

证明： 令1表示进栈，0表示出栈，则可转化为求一个2n位、含n个1、n个0的二进制数，满足从左往右扫描到任意一位时，

经过的0数不多于1数。显然含n个1、n个0的2n位二进制数共有 个，下面考虑不满足要求的数目.

考虑一个含n个1、n个0的2n位二进制数，扫描到第2m+1位上时有m+1个0和m个1（容易证明一定存在这样的情况），

则后面的0-1排列中必有n-m个1和n-m-1个0。将2m+2及其以后的部分0变成1、1变成0，

则对应一个n+1个0和n-1个1的二进制数。反之亦然（相似的思路证明两者一一对应）。 从而 。证毕。

Cn表示所有在n × n格点中不越过对角线的单调路径的个数。一个单调路径从格点左下角出发，

在格点右上角结束，每一步均为向上或向右。计算这种路径的个数等价于计算Dyck word的个数： X代表“向右”，Y代表“向上”。下图为n = 4的情况：

Cn表示通过连结顶点而将n + 2边的凸多边形分成三角形的方法个数。下图中为n = 4的情况：

Cn表示对{1, ..., n}依序进出栈的置换个数。一个置换w是依序进出栈的当S(w) = (1, ..., n),

其中S(w)递归定义如下：令w = unv，其中n为w的最大元素，u和v为更短1.括号化问题。

矩阵链乘： P=a1×a2×a3×……×an，依据乘法结合律，不改变其顺序，只用括号表示成对的乘积，

试问有几种括号化的方案？(h(n)种)

2.出栈次序问题。

一个栈(无穷大)的进栈序列为1,2,3,..n,有多少个不同的出栈序列?

类似： (1)有2n个人排成一行进入剧场。入场费5元。其中只有n个人有一张5元钞票，另外n人只有10元钞票，剧院无其它钞票，问有多少中方法使得只要有10元的人买票，售票处就有5元的钞票找零？(将持5元者到达视作将5元入栈，持10元者到达视作使栈中某5元出栈)

(2)在圆上选择2n个点,将这些点成对连接起来，使得所得到的n条线段不相交的方法数。

3.将多边行划分为三角形问题。

将一个凸多边形区域分成三角形区域的方法数?

类似：一位大城市的律师在她住所以北n个街区和以东n个街区处工作。每天她走2n个街区去上班。如果她

从不穿越（但可以碰到）从家到办公室的对角线，那么有多少条可能的道路？

类似：在圆上选择2n个点,将这些点成对连接起来使得所得到的n条线段不相交的方法数?

4.给顶节点组成二叉树的问题。

给定N个节点，能构成多少种形状不同的二叉树？

(一定是二叉树! 先去一个点作为顶点,然后左边依次可以取0至N-1个相对应的,右边是N-1到0个,

两两配对相乘,就是h(0)\*h(n-1) + h(2)\*h(n-2) + + h(n-1)h(0)=h(n)) （能构成h（N）个）

#define nmax 36

LL num[nmax];

int main() {

int i, cas, p, q, te;

num[0] = 1;

for (i = 1; i < nmax; i++) {

p = 4 \* i - 2, q = i + 1;

te = gcd(p, q);

p /= te, q /= te;

num[i] = num[i - 1] / q \* p;

}

cas = 0;

while (scanf("%d", &i) != EOF && (i != -1)) {

cas++;

printf("%d %d %I64d\n", cas, i, num[i] << 1);

}

return 0;

}

/\*

\* hdu 1023 1130 1134 2067

\*/

public class Main {

public static void main(String[] args) {

BigInteger[] num = new BigInteger[101];

BigInteger r, one, two;

int i;

one = BigInteger.*ONE*;

two = BigInteger.*valueOf*(2);

num[0] = BigInteger.*ONE*;

for (i = 1; i < 101; i++) {

r = BigInteger.*valueOf*(i);

num[i] = num[i - 1].multiply(two).multiply( two.multiply(r).subtract(one)).divide(r.add(one)); }

while (cin.hasNext()) {

i = cin.nextInt();

if (i == -1) {

break;

}

System.*out*.println(num[i]);

}}}

#define LL long long

#define nmax 200001

int prime[nmax], flag[nmax], factor[nmax], cfactor[nmax];

int x, y, plen;

void init() {

memset(flag, -1, sizeof(flag));

int i, j;

for (i = 2, plen = 0; i < nmax; i++) {

if (flag[i]) {

prime[plen++] = i;

}

for (j = 0; (j < plen) && (i \* prime[j] < nmax); j++) {

flag[i \* prime[j]] = 0;

if (i % prime[j] == 0) {

break;

}

}

}

}

int modular\_exp(int a, int b, int c) {

LL res, temp;

temp = a % c, res = 1;

while (b) {

if (b & 1) {

res = res \* temp % c;

}

b >>= 1;

temp = temp \* temp % c;

}

return res;

}

void extend\_gcd(int a, int b) {

if (b == 0) {

x = 1, y = 0;

return;

}

extend\_gcd(b, a % b);

int tx = x;

x = y;

y = tx - a / b \* y;

}

void solve(int n, int m) {

int i, j, k, temp, mm;

LL res, res0, res1;

mm = m;

temp = (int) (sqrt(m \* 1.0));

for (i = 0, k = 0; (i < plen) && (prime[i] <= temp); i++) {

if (mm % prime[i] == 0) {

factor[k++] = prime[i];

while (mm % prime[i] == 0) {

mm /= prime[i];

}

}

}

if (mm > 1) {

factor[k++] = mm;

}

memset(cfactor, 0, sizeof(cfactor));

for (i = 2, res0 = 1, res = 1; i <= n; i++) {

temp = 4 \* i - 2;

for (j = 0; j < k; j++) {

while (temp % factor[j] == 0) {

temp /= factor[j];

cfactor[j]++;

}

}

res0 = res0 \* temp % m;

temp = i + 1;

for (j = 0; j < k; j++) {

while (temp % factor[j] == 0) {

temp /= factor[j];

cfactor[j]--;

}

}

if (temp > 1) {

extend\_gcd(temp, m);

x = (x % m + m) % m;

res0 = res0 \* x % m;

}

res1 = res0;

for (j = 0; j < k; j++) {

if (cfactor[j] > 0) {

res1 = res1 \* modular\_exp(factor[j], cfactor[j], m) % m;

}

}

res = (res + res1) % m;

}

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

}

[Stirling (斯特灵数)的应用](http://www.cnblogs.com/xiaoxian1369/archive/2011/08/26/2154783.html)

小知识：

Bell数，又称为贝尔数。

是以埃里克·坦普尔·贝尔(Eric Temple Bell)为名的。

B(n)是包含n个元素的集合的划分方法的数目。

B(0) = 1, B(1) = 1, B(2) = 2, B(3) = 5,

B(4) = 15, B(5) = 52, B(6) = 203,...

递推公式为，

B(0) = 1,

B(n+1) = Sum(0,n) C(n,k)B(k). n = 1,2,...

其中，Sum(0,n)表示对k从0到n求和，C(n,k) = n!/[k!(n-k)!]

-------------------------

Stirling数，又称为斯特灵数。

在组合数学，Stirling数可指两类数，都是由18世纪数学家James Stirling提出的。

第一类Stirling数是有正负的，其绝对值是包含n个元素的集合分作k个环排列的方法数目。

递推公式为，

S(n,0) = 0, S(1,1) = 1.

S(n+1,k) = S(n,k-1) + nS(n,k)。

第二类Stirling数是把包含n个元素的集合划分为正好k个非空子集的方法的数目。

递推公式为，

S(n,n) = S(n,1) = 1,

S(n,k) = S(n-1,k-1) + kS(n-1,k).

将n个有区别的球的球放入k个无标号的盒子中( n>=k>=1，且盒子不允许为空)的方案数就是stirling数.(即含 n 个元素的集合划分为 k 个集合的情况数)

递推公式:　S(n,k) = 0 (k > n)　　S(n,1) = 1 (k = 1)

s(n,k)=1 (n=k) 　　S(n,k) = S(n-1,k-1)+k\*S(n-1,k) (n >= k >= 2)

　　分析：设有n个不同的球，分别用b1,b2,...,bn表示。从中取出一个球bn，bn的放法有以下两种：

　　1.bn独占一个盒子，那么剩下的球只能放在k-1个盒子里，方案数为S（n-1,k-1);

　　2.bn与别的球共占一个盒子，那么可以将b1,b2,...,bn-1这n-1个球放入k个盒子里，然后将bn放入其中一个盒子中，方案数为k\*S(n-1,m).

-------------

bell数和stirling数的关系为，

每个贝尔数都是"第二类Stirling数"的和。

B(n) = Sum(1,n) S(n,k).

hdu 2643 Rank hdu 2512 一卡通大冒险

hdu 2512

/\*

第二类Stirling数是把包含n个元素的集合划分为正好k个非空子集的方法的数目。

递推公式为：

S(n,k) = 0(n<k||k=0),

S(n,n) = S(n,1) = 1,

S(n,k) = S(n-1,k-1) + kS(n-1,k).

将n个有区别的球的球放入k个无标号的盒子中( n>=k>=1，且盒子不允许为空)的方案数就是stirling数.(即含 n 个元素的集合划分为 k 个集合的情况数)

　　递推公式:

　　S(n,k) = 0 (k > n)

　　S(n,1) = 1 (k = 1)

　　s(n,k)=1 (n=k)

　　S(n,k) = S(n-1,k-1)+k\*S(n-1,k) (n >= k >= 2)

　　分析：设有n个不同的球，分别用b1,b2,...,bn表示。从中取出一个球bn，bn的放法有以下两种：

　　1.bn独占一个盒子，那么剩下的球只能放在k-1个盒子里，方案数为S（n-1,k-1);

　　2.bn与别的球共占一个盒子，那么可以将b1,b2,...,bn-1这n-1个球放入k个盒子里，然后将bn放入其中一个盒子中，方案数为k\*S(n-1,m).

\*/

#define nmax 2001

#define nnum 1000

int num[nmax][nmax];

void init() {

int i, j;

for (i = 1; i < nmax; i++) {

num[i][0] = 0, num[i][1] = 1;

}

for (i = 2; i < nmax; i++) {

for (j = 1; j < nmax; j++) {

if (i == j) {

num[i][i] = 1;

continue;

}

num[i][j] = (num[i - 1][j - 1] + num[i - 1][j] \* j) % nnum;

}

}

}

int main() {

int n, x, i, res;

init();

while (scanf("%d", &n) != EOF) {

while (n--) {

scanf("%d", &x);

for (i = 1, res = 0; i <= x; i++) {

res += num[x][i];

res %= nnum;

}

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

}

}

return 0;

}

hdu 2643

/\* 第二类Stirling数是把包含n个元素的集合划分为正好k个非空子集的方法的数目。

 递推公式为：

 S(n,k) = 0(n<k||k=0),

 S(n,n) = S(n,1) = 1,

 S(n,k) = S(n-1,k-1) + kS(n-1,k).\*/

/\* 第二类Stirling数是把包含n个元素的集合划分为正好k个非空子集的方法的数目。

递推公式为： S(n,k) = 0(n<k||k=0),

S(n,n) = S(n,1) = 1,

S(n,k) = S(n-1,k-1) + kS(n-1,k).

\*/

#define LL long long

#define nmax 101

#define nnum 20090126LL

LL num[nmax][nmax], fac[nmax];

void init() {

int i, j;

for (i = 1, fac[0] = 1; i < nmax; i++) {

fac[i] = fac[i - 1] \* i % nnum;

}

for (i = 1; i < nmax; i++) {

num[i][1] = 1;

num[i][0] = 0;

}

for (i = 2; i < nmax; i++) {

for (j = 1; j < nmax; j++) {

if (i == j) {

num[i][i] = 1;

} else {

num[i][j] = (num[i - 1][j - 1] + num[i - 1][j] \* j) % nnum;

}

}

}

}

int main() {

init();

int T, N, i;

LL res;

while (scanf("%d", &T) != EOF) {

while (T--) {

scanf("%d", &N);

for (i = 1, res = 0; i <= N; i++) {

res += num[N][i] \* fac[i];

res %= nnum;

}

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

}

}

return 0;

}

hdu 3625 第一类斯特林数的应用

就是给你N个房间，然后每个房间1把钥匙，你最初手里没有任何钥匙，要靠破门而入！这里只有第一个房间不能破门进去，其他都可以，

给你房间数N，和最多能破门的个数，让你求能全部把房间打开的概率！

首先这题其实让我们求的是给 N个元素，让我们求K个环排列的 方法数。

斯特林第一类数的递推公式：

S（N，0）=0；

S（N，N）=1；

S（0，0）=0；

S（N，K）=S（N-1，K-1）+S（N-1，K）\*（N-1）；

这个公式的意思是：

当前N-1个数构成K-1 个环的时候，加入第N个 ，N只能构成单环！---S（N-1，K-1）

如果N-1个数构成K个环的时候，加入第N个，N可以任意加入，N-1内的一个环里，所以是--（N-1）\*S（N-1，K）

这个题目里，因为不能破坏第1个门：

所以 S（N，K）-S（N-1，K-1）才是能算构成K个环的方法数！就是去掉1自己成环的情况！

#define LL double

#define nmax 21

LL num[nmax], stirling[nmax][nmax];

void init() {

int i, j;

num[0] = 1;

for (i = 1; i < nmax; i++) {

num[i] = num[i - 1] \* i;

stirling[i][0] = 0;

stirling[i][1] = 1;

}

for (i = 2; i < nmax; i++) {

for (j = 1; j < nmax; j++) {

if (i == j) {

stirling[i][j] = 1;

} else {

stirling[i][j] = stirling[i - 1][j - 1]

+ (i - 1) \* stirling[i - 1][j];

}

}

}

}

LL solve(int n, int m) {

LL res;

int i;

for (i = 1, res = 0.0; i <= m; i++) {

res += stirling[n][i] - stirling[n - 1][i - 1];

}

return res;

}

int main() {

int T, N, K;

LL x;

double res;

init();

while (scanf("%d", &T) != EOF) {

while (T--) {

scanf("%d %d", &N, &K);

x = solve(N, K);

res = 1.0 \* x / num[N];

printf("%.4lf\n", res);

}

}

return 0;

}

斐波那契数列

1. 求解Fibonacci的某一项（这个范围一般在45之内) f[n]=f[n-1]+f[n-2] (n>2, f[1]=1,f[2]=1)

2. 求解Fibonacci的某一项模K(这个一般是大数),通用解决方法是构造矩阵求幂次

我比较喜欢下面这个形式

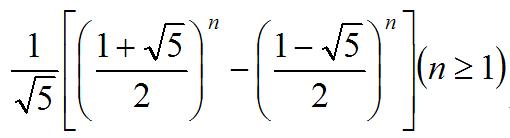
|0 1| |f(0)| |f(1)|

===>

|1 1| |f(1)| |f(0)+f(1)| 后面这个也就是f(2)了

3. 求解Fibonacci的前多少位 (这个一般是大数), 通用解法是使用通项公式

这个要使用fibonacci的通项公式了



可以发现后面的((1-√5)/2)^ n 当n很大的时候数值非常小，可以忽略，不会对前4位造成影响。

因此转变为了

((1+√5)/2 )^n

------------------

√5

4. 求解Fibonacci的后多少位，这个和取模类似

5. 求解Fibonacci的前n项和，利用推导式, S(n) = F(n + 2) - F(2)即可

F(3) = F(1) + F(2)

F(4) = F(2) + F(3) = 1 \* F(1) + 2 \* F(2)

F(5) = F(3) + F(4) = 2 \* F(1) + 3 \* F(2)

F(6) = F(4) + F(5) = 3 \* F(1) + 5 \* F(2)

F(7) = F(5) + F(6) = 5 \* F(1) + 8 \* F(2)

F(8) = F(6) + F(7) = 8 \* F(1) + 13 \* F(2)

S(3) = 2 \* F(1) + 2 \* F(2)

S(4) = 3 \* F(1) + 4 \* F(2)

S(5) = 5 \* F(1) + 7 \* F(2)

S(6) = 8 \* F(1) + 12 \*F(2)

S(7) = 13 \*F(1) + 20 \*F(2)

不难发现，

S(n) = F(n + 2) - F(2)

6. 更多的是基于Fibonacci的综合题,包括DP,构造,等等.

F(2\*n)=sigma(C(n,k)F(k)) k>=0,k<=n

母函数学习

普通母函数

题目：<http://acm.hdu.edu.cn/showproblem.php?pid=1028>

#define nmax 121

int res[nmax], temp[nmax];

int main() {

int n, i, j, k;

for (i = 0; i < nmax; i++) {

res[i] = 1;

temp[i] = 0;

}

for (i = 2; i < nmax; i++) {

for (j = 0; j < nmax; j++) {

for (k = 0; k + j < nmax; k += i) {

temp[k + j] += res[j];

}

}

for (j = 0; j < nmax; j++) {

res[j] = temp[j];

temp[j] = 0;

}

}

while (scanf("%d", &n) != EOF) {

printf("%d\n", res[n]);

}

return 0;

}

  题目：<http://acm.hdu.edu.cn/showproblem.php?pid=1171>

#define nmax 250001

int V[nmax], M[nmax], res[nmax], temp[nmax];

int main() {

int N, i, j, k, te, Vs;

while (scanf("%d", &N) != EOF && N > 0) {

for (i = 0, Vs = 0; i < N; i++) {

scanf("%d %d", V + i, M + i);

Vs += V[i] \* M[i];

}

te = Vs / 2;

memset(res, 0, sizeof(res));

memset(temp, 0, sizeof(temp));

for (i = 0; i <= M[0]; i++) {

res[i \* V[0]] = 1;

}

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

for (j = 0; j <= Vs; j++) {

for (k = 0; k <= M[i] && (V[i] \* k + j <= Vs); k++) {

if (res[j] == 1) {

temp[k \* V[i] + j] = 1;

}

}

}

for (j = 0; j <= Vs; j++) {

res[j] = temp[j], temp[j] = 0;

}

}

for (i = te; i >= 0; i--) {

if (res[i] != 0) {

break;

}

}

printf("%d %d\n", Vs - i, i);

}

return 0;

}

指数型母函数

//hdu1521

#define nmax 15

int num[nmax];

double res[nmax], fac[nmax], temp[nmax];

void init(int n) {

int i;

fac[0] = 1;

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

fac[i] = fac[i - 1] \* i;

}

}

int main() {

int i, j, k, n, m;

while (scanf("%d %d", &n, &m) != EOF) {

init(n);

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

scanf("%d", num + i);

}

memset(res, 0.0, sizeof(res));

memset(temp, 0.0, sizeof(res));

for (i = 0; i <= num[0]; i++) {

res[i] = 1.0 / fac[i];

}

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

for (j = 0; j <= n; j++) {

for (k = 0; (k <= num[i]) && (k + j <= n); k++) {

temp[j + k] += res[j] / fac[k];

}

}

for (j = 0; j <= n; j++) {

res[j] = temp[j], temp[j] = 0;

}

}

printf("%.0lf\n", res[m] \* fac[m]);

}

return 0;

}

//poj 3734 hdu 2065

/\*由题知：

(1+x/1!+x^2/2!+……+x^n/n!)^2\*(1+x^2/2!+……)^2

由e^x=1+x/1!+x^2/2!+……知

原式=e^(2\*x)\*((e^x+e^(-x))/2)^2

　　=(1/4)\*(e^(2\*x)+1)^2

　　=(1/4)\*(e^(4\*x)+2\*e^(2\*x)+1)

　　=(1/4)\*(sigam(4^n)\*(x^n/n!)+

2\*sigma(2^n)\*(x^n/n!)+1)

由以上式子可知：　x^n/n!的系数为(4^n+2\*2^n+1)/4=4^(n-1)+2^(n-1)+1/4

对于本题只需要计算4^(n-1)+2^(n-1)即可。

\*/

Polya定理

利用置换论中轮换的概念（轮换概念=循环概念），我们将置换f分解为若干循环。记f的循环节数位m(f)。

则有定理:C(f) = k ^ m(f) ，其中看为着色数。

于是有Polya定理：不同的着色方案数 l = sum(k ^ m(f)) / |G| ，其中f属于G。

对于本题还有如下结论：

在此种模型下，所有的置换可以通过一个置换旋转和翻转全部得来。

（1）旋转

将置换顺时针旋转i格，其循环节数的为gcd(n, i)；

（2）翻转

当n为奇数：共有n个循环节数为(n+1)/2的循环群

当 n为偶数：共有n/2个循环节数(n+2)/2的循环群，和n/2个循环节数n/2的循环群。

#define nnum 1000000007

LL polya(int n, int m) {

int i;

LL sum;

for (i = 1, sum = 0; i <= m; i++) {

if (m % i == 0) {

sum += modular\_exp(n, i) \* getPhi(m / i) % nnum;

sum %= nnum;

}

}

if (m & 1) {

sum += modular\_exp(n, (m + 1) / 2) \* m % nnum;

sum %= nnum;

} else {

sum += modular\_exp(n, m / 2) \* (m / 2) % nnum;

sum %= nnum;

sum += modular\_exp(n, m / 2 + 1) \* (m / 2) % nnum;

sum %= nnum;

}

extend\_gcd(m << 1, nnum);

x = (x % nnum + nnum) % nnum;

sum = sum \* x % nnum;

return sum;

}

dfs实现容斥原理计数

/\*\*

\* 容斥原理计数的例子。给定区间[1,N)和M个数(data[])

\* 求区间中有多少个数满足至少被这些数中的一个整除

\* 省略了输入过程,调用work即得结果。

\*/

const int MAXM = 15;

int N, M, data[MAXM];

void dfs(int now, int count, LL lcm, LL &ans) {

lcm = getlcm(lcm, data[now]);

if(count % 2 == 0) {

ans -= (N - 1) / lcm;

}else {

ans += (N - 1) / lcm;

}

for(int i = now + 1; i < M; i++) {

dfs(i, count + 1, lcm, ans);

}

}

LL work(int N, int M) {

LL ans = 0;

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

dfs(i, 1, data[i], ans);

}

return ans;

}

排列组合生成

//gen\_perm产生字典序排列P(n,m)

//gen\_comb产生字典序组合C(n,m)

//gen\_perm\_swap产生相邻位对换全排列P(n,n)

//产生元素用1..n表示

//dummy为产生后调用的函数,传入a[]和n,a[0]..a[n-1]为一次产生的结果

#define MAXN 100

int count;

void dummy(int\* a, int n) {

int i;

cout << count++ << ": ";

for (i = 0; i < n - 1; i++)

cout << a[i] << ' ';

cout << a[n - 1] << endl;

}

void \_gen\_perm(int\* a, int n, int m, int l, int\* temp, int\* tag) {

int i;

if (l == m)

dummy(temp, m);

else

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

if (!tag[i]) {

temp[l] = a[i], tag[i] = 1;

\_gen\_perm(a, n, m, l + 1, temp, tag);

tag[i] = 0;

}

}

void gen\_perm(int n, int m) {

int a[MAXN], temp[MAXN], tag[MAXN] = { 0 }, i;

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

a[i] = i + 1;

\_gen\_perm(a, n, m, 0, temp, tag);

}

void \_gen\_comb(int\* a, int s, int e, int m, int& count, int\* temp) {

int i;

if (!m)

dummy(temp, count);

else

for (i = s; i <= e - m + 1; i++) {

temp[count++] = a[i];

\_gen\_comb(a, i + 1, e, m - 1, count, temp);

count--;

}

}

void gen\_comb(int n, int m) {

int a[MAXN], temp[MAXN], count = 0, i;

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

a[i] = i + 1;

\_gen\_comb(a, 0, n - 1, m, count, temp);

}

void \_gen\_perm\_swap(int\* a, int n, int l, int\* pos, int\* dir) {

int i, p1, p2, t;

if (l == n)

dummy(a, n);

else {

\_gen\_perm\_swap(a, n, l + 1, pos, dir);

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

p2 = (p1 = pos[l]) + dir[l];

t = a[p1], a[p1] = a[p2], a[p2] = t;

pos[a[p1] - 1] = p1, pos[a[p2] - 1] = p2;

\_gen\_perm\_swap(a, n, l + 1, pos, dir);

}

dir[l] = -dir[l];

}

}

void gen\_perm\_swap(int n) {

int a[MAXN], pos[MAXN], dir[MAXN], i;

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

a[i] = i + 1, pos[i] = i, dir[i] = -1;

\_gen\_perm\_swap(a, n, 0, pos, dir);

}

生成gray码

//生成reflected gray code

//每次调用gray取得下一个码

//000...000是第一个码,100...000是最后一个码

void gray(int n, int \*code) {

int t = 0, i;

for (i = 0; i < n; t += code[i++])

;

if (t & 1)

for (n--; !code[n]; n--)

;

code[n - 1] = 1 - code[n - 1];

}

置换(polya)

//求置换的循环节,polya原理

//perm[0..n-1]为0..n-1的一个置换(排列)

//返回置换最小周期,num返回循环节个数

#define MAXN 1000

int gcd(int a, int b) {

return b ? gcd(b, a % b) : a;

}

int polya(int\* perm, int n, int& num) {

int i, j, p, v[MAXN] = { 0 }, ret = 1;

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

if (!v[i]) {

for (num++, j = 0, p = i; !v[p = perm[p]]; j++)

v[p] = 1;

ret \*= j / gcd(ret, j);

}

return ret;

}

字典序全排列

//字典序全排列与序号的转换

int perm2num(int n, int \*p) {

int i, j, ret = 0, k = 1;

for (i = n - 2; i >= 0; k \*= n - (i--))

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

if (p[j] < p[i])

ret += k;

return ret;

}

void num2perm(int n, int \*p, int t) {

int i, j;

for (i = n - 1; i >= 0; i--)

p[i] = t % (n - i), t /= n - i;

for (i = n - 1; i; i--)

for (j = i - 1; j >= 0; j--)

if (p[j] <= p[i])

p[i]++;

}

字典序组合

//字典序组合与序号的转换

//comb为组合数C(n,m),必要时换成大数,注意处理C(n,m)=0|n<m

int comb(int n, int m) {

int ret = 1, i;

m = m < (n - m) ? m : (n - m);

for (i = n - m + 1; i <= n; ret \*= (i++))

;

for (i = 1; i <= m; ret /= (i++))

;

return m < 0 ? 0 : ret;

}

int comb2num(int n, int m, int \*c) {

int ret = comb(n, m), i;

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

ret -= comb(n - c[i], m - i);

return ret;

}

void num2comb(int n, int m, int\* c, int t) {

int i, j = 1, k;

for (i = 0; i < m; c[i++] = j++)

for (; t > (k = comb(n - j, m - i - 1)); t -= k, j++)

;

}

1. 字符串、匹配

Trie树

/\*\*

\* Trie树(单词前缀树)比较通用的实现，以下代码实现Trie树最

\* 基本的功能，如插入字符串等。插入完毕后可根据具体需要对

\* 树进行操作，如数结点数，查找字符串等。以下代码默认是采

\* 用静态分配内存的方式，树根即为root。若改为动态方式，

\* 则需要new，并在使用完毕后destroy。

\*/

/\*\*

\* 子节点个数最大值，例如如果全是小写字母的话，

\* 则chileNum应该为26

\*/

const int child\_num = 26;

/\*\*

\* 参照字符，例如如果全是小写字母的话，则base应该为'a'

\*/

const char base = 'a';

/\*\*

\* 节点结构体

\* 可在其中加入自定义变量以满足特殊需求

\*/

typedef struct TrieNode {

//子节点指针

struct TrieNode \*child[child\_num];

}TrieNode;

/\*\*

\* 以下三行代码定义了节点数组，以及变量指示当前已经分配

\* 节点数量，用于静态地给新建结点分配内存空间如果新建结

\* 点时采用动态分配内存的方式，则可删除。

\*/

const int MAX\_NODE\_NUM = 300000;

TrieNode root[MAX\_NODE\_NUM];

int cur\_node\_num = 0;

/\*\*

\* 新建一个节点的函数，新建后会自动初始化

\*/

inline TrieNode\* newTrieNode() {

//如需动态分配内存，则下句可改为

//TrieNode\* node = new TrieNode();

TrieNode\* node = &root[cur\_node\_num++];

//初始化

memset(node->child, 0, sizeof(node->child));

return node;

}

/\*\*

\* 插入一个字符串到Trie树中,root为要插入的位置

\* s为字符串，len为其长度

\*/

void insert\_to\_trie(TrieNode \*root, const char \*s, const int len) {

TrieNode \*p = root;

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

if (p->child[s[i] - base] == NULL) {

p->child[s[i] - base] = newTrieNode();

}

p = p->child[s[i] - base];

}

}

/\*\*

\* 如果建立Trie树过程中使用的是动态分配内存的方式，

\* 则使用完毕以后需要摧毁，以及时释放占用的内存

\*/

void destroy\_node(const TrieNode \*p) {

if (p) {

for (int i = 0; i < child\_num; i++) {

if (p->child[i] != NULL) {

destroy\_node(p->child[i]);

}

}

delete p;

}

}

KMP

//经典的字符串匹配算法，复杂度O(m+n)。使用时以下get\_next()函数与get\_nextval()函数只需要一个即可，调用的地方也要相应修改。get\_nextval()是在get\_next()朴素算法的基础上进一步优化

#define MAX\_PAR\_LEN 300

#define MAX\_TXT\_LEN 1000

char pattern[MAX\_PAR\_LEN], text[MAX\_TXT\_LEN];

int next[MAX\_PAR\_LEN], nextval[MAX\_PAR\_LEN], parlen, txtlen;

void get\_next() {

int i = 0, j = -1; next[0] = -1;

while (i < parlen) {

if (j < 0 || pattern[i] == pattern[j]) {

i++; j++; next[i] = j;

} else { j = next[j]; }

}

}

void get\_nextval() {

int i = 0, j = -1; nextval[0] = -1;

while (i < parlen) {

if (j < 0 || pattern[i] == pattern[j]) {

i++; j++;

if (pattern[i] != pattern[j]) {

nextval[i] = j;

} else { nextval[i] = nextval[j]; }

} else { j = nextval[j];}

}

}

int index\_kmp() {

int i, j;

txtlen = strlen(text);

parlen = strlen(pattern);

get\_nextval();

i = j = 0;

while (i < txtlen && j < parlen) {

if (j < 0 || text[i] == pattern[j]) {

i++; j++;

} else { j = nextval[j];}

}

if (j == parlen) { return i - j;

} else { return -1;}

}

KMP实现DFA(单模式串)

#define MAX\_LEN 10005

char pattern[MAX\_LEN];

int state[MAX\_LEN][26];

int len;

int nextval[MAX\_LEN];

void get\_next() {

int i = 0, j = -1;

int parlen = len;

nextval[0] = -1;

while (i <= parlen) {

if (j == -1 || pattern[i] == pattern[j]) {

i++;

j++;

nextval[i] = j;

} else {

j = nextval[j];

}

}

}

void work() {

int i, j, cur;

char c;

while (scanf("%s", pattern) != EOF) {

if (strcmp(pattern, "0") == 0) {

break;

}

len = strlen(pattern);

get\_next();

memset(state, 0, sizeof(state));

for (cur = 0; cur <= len; cur++) {

for (c = 'a'; c <= 'z'; c++) {

i = cur;

while(i >= 0 && c != pattern[i]) {

i = nextval[i];

}

state[cur][c - 'a'] = i + 1;

}

}

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

printf("%d", i);

for (j = 0; j < 26; j++) {

printf(" %d", state[i][j]);

}

printf("\n");

}

}

}

Trie树实现DFA(多模式串)

const int MAX\_PATTERN\_NUM = 510;

const int MAX\_PATTERN\_LEN = 210;

const int MAXQ = MAX\_PATTERN\_NUM \* MAX\_PATTERN\_LEN;

const int MAX\_TEXT\_LEN = 10100;

const int MAXK = 95; //字符集的大小

const char BASE = '!';

typedef struct TrieNode {

TrieNode\* fail;

TrieNode\* next[MAXK];

bool isaend; //该节点是否为某模式串的终结点

int index; //以该节点为终结点的模式串编号

} TrieNode;

TrieNode \*que[MAXQ], \*root, CTree[MAXQ];

//文本字符串及模式串

char msg[MAX\_TEXT\_LEN];

char pattern[MAX\_PATTERN\_LEN];

int N, nCount;

bool infected[MAX\_PATTERN\_NUM];

inline TrieNode \* NewTrieNode() {

CTree[nCount].index = -1;

return &CTree[nCount++];

}

void TrieInsert(char \*s, int index) {

int i = 0;

TrieNode \*ptr = root;

while (s[i]) {

int idx = s[i] - BASE;

if (!ptr->next[idx]) {

ptr->next[idx] = NewTrieNode();

}

ptr = ptr->next[idx];

i++;

}

ptr->isaend = true;

ptr->index = index;

}

void Init() {

int i;

root = NewTrieNode();

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

scanf("%s", pattern);

TrieInsert(pattern, i);

}

}

void Build\_DFA() {

int rear = 1, front = 0, i;

que[0] = root;

root->fail = NULL;

while (rear != front) {

TrieNode \*cur = que[front++];

for (i = 0; i < MAXK; i++) {

if (!cur->next[i]) {

continue;

}

if (cur == root) {

cur->next[i]->fail = root;

} else {

TrieNode \*ptr = cur->fail;

while (ptr) {

if (ptr->next[i]) {

cur->next[i]->fail = ptr->next[i];

if (ptr->next[i]->isaend) {

cur->next[i]->isaend = true;

}

break;

}

ptr = ptr->fail;

}

if (!ptr) {

cur->next[i]->fail = root;

}

}

que[rear++] = cur->next[i];

}

}

}

void Run\_DFA() {

int i = 0;

TrieNode \*ptr = root;

ptr = root;

while (msg[i]) {

int idx = msg[i] - BASE;

while (!ptr->next[idx] && ptr != root) {

ptr = ptr->fail;

}

ptr = ptr->next[idx];

if (!ptr) {

ptr = root;

}

TrieNode \*tmp = ptr;

while (tmp && tmp->isaend) {

infected[tmp->index] = true;

tmp = tmp->fail;

}

i++;

}

}

int main() {

int M, total;

while (scanf("%d", &N) == 1) {

nCount = 0;

total = 0;

memset(CTree, 0, sizeof(CTree));

Init();

Build\_DFA();

scanf("%d", &M);

for (int i = 1; i <= M; i++) {

memset(infected, false, sizeof(infected));

scanf("%s", msg);

Run\_DFA();

int temp = 0;

for (int j = 1; j <= N; j++) {

temp += infected[j];

}

if (temp > 0) {

total++;

printf("web %d:", i);

for (int j = 1; j <= N; j++) {

if (infected[j]) {

printf(" %d", j);

}

}

putchar('\n');

}

}

printf("total: %d\n", total);

}

return 0;

}

AC自动机（DFA）

/\*

\* 初始化：

\* \*root = new node();

\* insert()加入所有串后调用build\_ac\_automation(node \*root)

\* 调用：query(原串)

\* 返回：匹配成功的单词数量

\* 调用：

\*/

const int kind = 26;

struct node

{

node \*fail; //失败指针

node \*next[kind]; //Tire每个节点的26个子节点（最多26个字母）

int count; //是否为该单词的最后一个节点

node()

{ //构造函数初始化

fail = NULL;

count = 0;

memset(next, (int) NULL, sizeof(next));

}

}\*q[500001] , \*root; //队列，方便用于bfs构造失败指针

void insert(char \*str, node \*root)

{

node \*p = root;

int i = 0, index;

while (str[i])

{

index = str[i] - 'a';

if (p->next[index] == NULL)

p->next[index] = new node();

p = p->next[index];

i++;

}

p->count++;

}

void build\_ac\_automation(node \*root)

{

int i;

int head, tail; //队列的头尾指针

head = tail = 0;

root->fail = NULL;

q[head++] = root;

while (head != tail)

{

node \*temp = q[tail++];

node \*p = NULL;

for (i = 0; i < 26; i++)

{

if (temp->next[i] != NULL)

{

p = temp->fail;

while (p != NULL && p->next[i] == NULL)

p = p->fail;

if (p == NULL)

temp->next[i]->fail = root;

else

temp->next[i]->fail = p->next[i];

q[head++] = temp->next[i];

}

}

}

}

int query(node \*root, char\* str)

{

int i = 0, cnt = 0, index;

node \*p = root;

while (str[i])

{

index = str[i] - 'a';

while (p->next[index] == NULL && p != root)

p = p->fail;

p = p->next[index];

p = (p == NULL) ? root : p;

node \*temp = p;

while (temp != root && temp->count != -1)

{

cnt += temp->count;

temp->count = -1;//若每个单词可以匹配多次则去掉

temp = temp->fail;

}

i++;

}

return cnt;

}

Sunday算法

#define CHNUM 26

#define BASE 'a'

int sunday(const char \*src, const char \*des) {

int i, j, pos = 0;

int len\_s, len\_d;

int next[CHNUM] = { 0 };

len\_s = strlen(src);

len\_d = strlen(des);

for (j = 0; j < CHNUM; ++j) {

next[j] = len\_d + 1;

}

for (j = 0; j < len\_d; ++j) {

next[des[j] - BASE] = len\_d - j;

}

while (pos < (len\_s - len\_d + 1)) {

i = pos;

for (j = 0; j < len\_d; ++j, ++i) {

if (src[i] != des[j]) {

pos += next[src[pos + len\_d] - BASE];

break;

}

}

if (j == len\_d) {

return pos;

}

}

return -1;

}

1. 数据结构

RMQ

/\*

\* 初始化：InitRMQ()

\* 调用：query(x,y,st)求区间[x,y]的最大值

\* 返回：下标最小的最大值的下标

\*/

int d[30];

void InitRMQ(int f[], int n, int st[][32])

{

int i, j;

for (d[0] = 1, i = 1; i < 21; i++)

d[i] = 2 \* d[i - 1];

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

st[i][0] = i;

int k = int(log(double(n))/log(2)) + 1;

for (j = 1; j < k; j++)

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

if (i + d[j - 1] - 1 < n)

{

if (f[st[i][j - 1]] < f[st[i + d[j - 1]][j – 1]])//求最小值需改为大于号，若要下标最大则需将“<”或者“>”改为为“<=”或者“>=”

st[i][j] = st[i + d[j - 1]][j - 1];

else

st[i][j] = st[i][j - 1];

}

else

break;

}

int query(int x, int y, int st[][32])

{

int k = int(log(double(y - x + 1)) / log(2.0));

if (f[st[x][k]] < f[st[y - d[k] + 1][k]])//求最小值需改为大于号，若要下标最大则需将“<”或者“>”改为为“<=”或者“>=”

return st[y - d[k] + 1][k];

return st[x][k];

}

一维线段树

//区间最大值最小值之差，题目见北大OJ-3264

#define MY\_MIN 99999999

#define MY\_MAX -99999999

typedef struct CNode {

int L, R; //区间起点和终点

int nMin, nMax; //本区间里的最大最小值

CNode \*pLeft, \*pRight;

} CNode;

CNode Tree[500000];

int N, Q;

int nCount;

int CurMin, CurMax;

inline int min(int first, int second){

return first < second ? first : second;

}

inline int max(int first, int second){

return first > second ? first : second;

}

void BuildTree(CNode \*pRoot, int L, int R) {//建立线段树

pRoot->nMax = MY\_MAX;

pRoot->nMin = MY\_MIN;

pRoot->L = L;

pRoot->R = R;

if (L == R) {

return;

}

nCount++;

pRoot->pLeft = Tree + nCount;

nCount++;

pRoot->pRight = Tree + nCount;

int mid = (L + R) / 2;

BuildTree(pRoot->pLeft, L, mid);

BuildTree(pRoot->pRight, mid + 1, R);

}

void Insert(CNode \*pRoot, int i, int v) {//插入数据

pRoot->nMax = max(pRoot->nMax, v);

pRoot->nMin = min(pRoot->nMin, v);

if(pRoot->L == pRoot->R){

return ;

}

int mid = (pRoot->L + pRoot->R) / 2;

if(i <= mid)

Insert(pRoot->pLeft, i, v);

else

Insert(pRoot->pRight, i, v);

}

void Query(CNode \*pRoot, int s, int e){

if(pRoot->nMax <= CurMax && pRoot->nMin >= CurMin){

return ;

}

if(s == pRoot->L && e == pRoot->R){

CurMax = max(CurMax, pRoot->nMax);

CurMin = min(CurMin, pRoot->nMin);

return ;

}

int mid = (pRoot->L + pRoot->R) / 2;

if(e <= mid){

Query(pRoot->pLeft, s, e);

}

else if(s > mid){

Query(pRoot->pRight, s, e);

}

else{

Query(pRoot->pLeft, s, mid);

Query(pRoot->pRight, mid + 1, e);

}

}

int main() {

int h, i, s, e;

scanf("%d%d", &N, &Q);

nCount = 0;

BuildTree(Tree, 1, N);

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

scanf("%d", &h);

Insert(Tree, i, h);

}

for(i = 0; i < Q; i++){

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

CurMin = MY\_MIN;

CurMax = MY\_MAX;

Query(Tree, s, e);

printf("%d\n", CurMax - CurMin);

}

return 0;

}

快速排序

/\*

\* 调用：quick\_sort(f,l,r)分别传入数组名，最左和最右元素的下标

\* 注意：这是从小到大排序，若想改为从大到小，需要修改的地方已经在注释中标出。

\*/

void quick\_sort(int f[], int l, int r)

{

int i = l;

int j = r;

int x = f[l];

while (i < j)

{

while (i < j && f[j] >= x)//f[j]<=x

j--;

f[i] = f[j];

while (i < j && f[i] <= x)//f[i]>=x

i++;

f[j] = f[i];

}

f[i] = x;

if (l < i - 1)

quick\_sort(f, l, i - 1);

if (i + 1 < r)

quick\_sort(f, i + 1, r);

}

平面上矩形面积并

typedef struct CNode {

int L, R;

int covers;

double lastx;

CNode \*pLeft, \*pRight;

} CNode;

typedef struct {

double x;

double yi;

double ya;

int cover;

} Line;

const int MAXN = 205;

const int MAX\_NODE = 4 \* MAXN;

CNode Tree[MAX\_NODE];

Line lines[2 \* MAXN];

double ys[2 \* MAXN], ans;

int N, N2, nCount;

inline bool operator<(const Line &a, const Line &b) {

return a.x < b.x;

}

//建立线段树

void BuildTree(CNode \*pRoot, int L, int R) {

pRoot->L = L;

pRoot->R = R;

pRoot->covers = 0;

pRoot->lastx = 0;

if (L == R) {

return;

}

nCount++;

pRoot->pLeft = Tree + nCount;

nCount++;

pRoot->pRight = Tree + nCount;

int mid = (L + R) / 2;

BuildTree(pRoot->pLeft, L, mid);

BuildTree(pRoot->pRight, mid + 1, R);

}

//插入数据

void Update(CNode \*pRoot, double i, double v, double x, int cover) {

if (pRoot->L == pRoot->R) {

if (pRoot->covers) {

ans += (ys[pRoot->R] - ys[pRoot->L - 1]) \* (x - pRoot->lastx);

}

pRoot->covers += cover;

pRoot->lastx = x;

return;

}

int mid = (pRoot->L + pRoot->R) / 2;

if (pRoot->covers) {

Update(pRoot->pLeft, ys[pRoot->L - 1], ys[mid], pRoot->lastx,

pRoot->covers);

Update(pRoot->pRight, ys[mid], ys[pRoot->R], pRoot->lastx,

pRoot->covers);

pRoot->covers = 0;

}

if (ys[mid] > i && ys[mid] < v) {

Update(pRoot->pLeft, i, ys[mid], x, cover);

Update(pRoot->pRight, ys[mid], v, x, cover);

} else if (ys[mid] >= v) {

Update(pRoot->pLeft, i, v, x, cover);

} else {

Update(pRoot->pRight, i, v, x, cover);

}

}

void work() {

int T = 0;

double xi, yi, xa, ya;

while (scanf("%d", &N) == 1 && N > 0) {

N2 = 0;

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

scanf("%lf%lf%lf%lf", &xi, &yi, &xa, &ya);

lines[N2].yi = yi;

lines[N2].ya = ya;

lines[N2].cover = 1;

lines[N2].x = xi;

ys[N2++] = yi;

lines[N2].yi = yi;

lines[N2].ya = ya;

lines[N2].cover = -1;

lines[N2].x = xa;

ys[N2++] = ya;

}

nCount = 0;

BuildTree(Tree, 1, N2 - 1);

sort(ys, ys + N2);

sort(lines, lines + N2);

ans = 0;

for (int i = 0; i < N2; i++) {

Update(Tree, lines[i].yi, lines[i].ya, lines[i].x, lines[i].cover);

}

printf("Test case #%d\nTotal explored area: %.2f\n\n", ++T, ans);

}

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("data.in", "r", stdin);

#endif

work();

return 0;

}

一维树状数组

/\*\*

\* 一维树状树组，多用于区间求和。能在log(n)级的时间内求和、

\* 更新。注意：本段代码数组下标是从1开始的。

\*/

const int MAXN = 50100;

typedef int typec;

int N;//数据个数

typec data[MAXN];//存储数据，下标从1开始

inline int lowbit(const int &x) {//x > 0

return x & (-x);

}

//查询data[1...pos]的和

typec sum(int pos) {

typec ret = 0;

for(int i = pos; i > 0; i -= lowbit(i)) {

ret += data[i];

}

return ret;

}

//修改data[pos]的值，在原来基础上加value

void modify(int pos, typec value) {

for (int i = pos; i <= N; i += lowbit(i)) {

data[i] += value;

}

}

二维树状数组

/\*\*

\* 二维树状树组，多用于二维区域求和。能在log(n)级时间内求和、

\* 更新。用法与前面的一维树状数组类似。数组下标从1开始。

\*/

typedef long long LL;

const int MAXN = 1005;

LL data[MAXN][MAXN];

int Row = 1001, Col = 1001;

inline int Lowbit(const int &x) { // x > 0

return x & (-x);

}

LL sum(int x, int y) {

LL ret = 0;

for(int i = x; i > 0; i -= Lowbit(i)) {

for(int j = y; j > 0; j -= Lowbit(j)) {

ret += data[i][j];

}

}

return ret;

}

void update(int x, int y, int delta) {

for(int i = x; i <= Row; i += Lowbit(i)) {

for(int j = y; j <= Col; j += Lowbit(j)) {

data[i][j] += delta;

}

}

}

1. 动态规划

最长公共子序列长度

char str1[SIZE], str2[SIZE];

int c[2][SIZE];

void LCS(int m, int n) {

int i, j;

memset(c[0], 0, sizeof(int) \* SIZE);

c[1][0] = 0;

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

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

if (str1[i - 1] == str2[j - 1]) {

c[i & 1][j] = c[~i & 1][j - 1] + 1;

} else if (c[~i & 1][j] >= c[i & 1][j - 1]) {

c[i & 1][j] = c[~i & 1][j];

} else {

c[i & 1][j] = c[i & 1][j - 1];

}

}

}

printf("%d\n", c[m & 1][n]);

}

数位DP:1-n包含49的个数

typedef long long I64;

typedef unsigned long long U64;

/\*

\* dp[i][0]表示长度为i,不包含49的数字的个数

\* dp[i][1]表示长度为i，不包含49但以9开头的数字的个数

\* dp[i][2]表示长度为i，包含49的数字的个数

\*/

U64 dp[21][3];

char digit[25];

void init() {

dp[0][0] = 1;

dp[0][1] = 0;

dp[0][2] = 0;

for (int i = 1; i < 20; i++) {

dp[i][0] = dp[i - 1][0] \* 10 - dp[i - 1][1];

dp[i][1] = dp[i - 1][0];

dp[i][2] = dp[i - 1][2] \* 10 + dp[i - 1][1];

}

}

void work() {

int T, len;

U64 N, ans;

bool flag;

init();

scanf("%d", &T);

while (T--) {

scanf("%I64u", &N);

N++; //加1便于处理形如..49的数

sprintf(digit, "%I64d", N);

len = strlen(digit);

ans = 0;

flag = false;

for (int i = len; i > 0; i--) {

ans += dp[i - 1][2] \* (digit[len - i] - '0');

if (flag) {

ans += dp[i - 1][0] \* (digit[len - i] - '0');

} else {

if (digit[len - i] > '4') {

ans += dp[i - 1][1];

}

}

if (i < len && digit[len - i - 1] == '4' && digit[len - i] == '9') {

flag = true;

}

}

printf("%I64u\n", ans);

}

}

1. 网络流

匈牙利算法(邻接矩阵)

/\*求二分图最大匹配的经典算法。先调用buildgraph建图，然后调

\*用hungary，返回最大匹配数。深搜+邻接阵实现，复杂度O(N^3)。

\*N和M分别表示二分图两个部分的顶点数。mymap用于存储二分\*图，mymap充分利用了二分图的特性，不是通常的邻接矩阵，\*mymap[a][b]=true表示第一部分的顶点a与第二部分的顶点b有边

\*/

const int MAXN = 555;

int N, M, mymatch[MAXN];

bool visited[MAXN], mymap[MAXN][MAXN];

bool dfs(int k) {

int t;

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

if (mymap[k][i] && !visited[i]) {

visited[i] = true; t = mymatch[i]; mymatch[i] = k;

if (t == -1 || dfs(t)) {

return true;

}

mymatch[i] = t;

}

}

return false;

}

bool buildgraph(int K) {

int t1, t2;

memset(mymap, false, sizeof(mymap));

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

scanf("%d%d", &t1, &t2);

mymap[t1 - 1][t2 - 1] = true;

}

return true;

}

int hungary () {

memset(mymatch, -1, sizeof(mymatch));

int ans = 0;

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

memset(visited, false, sizeof(visited));

if (dfs(i)) { ans++;}

}

return ans;

}

匈牙利算法(邻接表)

/\*求二分图最大匹配的经典算法。先调用buildgraph建图，然后调

\*用hungary，返回最大匹配数。深搜+邻接表实现，复杂度O(N\*V)，\*适合稀疏图。

\*/

const int MAXN = 555;

vector<int> mymap[MAXN];

int N, M, mymatch[MAXN];

bool visited[MAXN];

void init() {

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

mymap[i].clear();

}

}

bool dfs(int k) {

int t, I;

for(int i = 0; i < (int)mymap[k].size(); i++) {

I = mymap[k][i];

if(!visited[I]) {

visited[I] = true; t = mymatch[I]; mymatch[I] = k;

if(t == -1 || dfs(t)) {

return true;

}

mymatch[I] = t;

}

}

return false;

}

bool buildgraph(int K) {

int t1, t2;

init();

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

scanf("%d%d", &t1, &t2);

mymap[t1 - 1].push\_back(t2 - 1);

}

return true;

}

int hungary () {

memset(mymatch, -1, sizeof(mymatch));

int ans = 0;

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

memset(visited, false, sizeof(visited));

if (dfs(i)) {

ans++;

}

}

return ans;

}

最小点集覆盖

重要结论：一个二分图的最小点集覆盖数等于其最大匹配数

KM算法

/\*

\* 初始化：w[x][y]为邻接矩阵。不存在的边赋值为-INF

\* 调用：KM(n,m,ans)n,m分别为x,y集合中的点数，n必须小于等于m。ans为回传结果。

\* 返回：false表示无完备匹配，true表示有完备匹配，且最佳匹配值为ans。

\*/

#define INF 0x3f3f3f3f

#define maxn 205

int w[maxn][maxn];

int lx[maxn], ly[maxn];

int linky[maxn];

int visx[maxn], visy[maxn];

int lack;

int m, n;

bool find(int x, int n, int m)

{

visx[x] = true;

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

{

if (visy[y])

continue;

int t = lx[x] + ly[y] - w[x][y];

if (t > 0)

{

lack = min(lack, t);

continue;

}

visy[y] = true;

if (linky[y] == -1 || find(linky[y], n, m))

{

linky[y] = x;

return true;

}

}

return false;

}

bool KM(int n, int m, int &res)

{

memset(linky, -1, sizeof(linky));

memset(lx, 0, sizeof(lx));

memset(ly, 0, sizeof(ly));

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

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

if (w[i][j] > lx[i])

lx[i] = w[i][j]; //初始化顶标

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

{

while (true)

{

memset(visx, 0, sizeof(visx));

memset(visy, 0, sizeof(visy));

lack = INF;

if (find(x, n, m))

break;

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

if (visx[i])

lx[i] -= lack;

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

if (visy[i])

ly[i] += lack;

}

}

res = 0;

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

{

if (w[linky[i]][i] <= -INF)

return false;

res += w[linky[i]][i];

}

return true;

}

最小费用最大流

/\*

\* 初始化：ncount=0;

\* memset(head, -1, sizeof(head));

\* addedge()加入所有弧，要加反向边

\* 调用：mincost()

\* 返回：最小费用，flow中存储流量。

\*/

struct Edge

{

int next, v, c, w;

}edge[E];

int nv, ncount;

bool vis[N];

int q[N], head[N], pv[N], pe[N];

int flow;

int cost, d[N];

int src, sink;

void addedge(int u, int v, int c, int w)

{

edge[ncount].v = v;

edge[ncount].c = c;

edge[ncount].w = w;

edge[ncount].next = head[u];

head[u] = ncount++;

}

int mincost(int src, int sink)

{

int front, rear;

int mxf;

flow = 0;

cost = 0;

while (true)

{

memset(pv, -1, sizeof(pv));

memset(vis, 0, sizeof(vis));

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

d[i] = inf;

d[src] = 0;

pv[src] = src;

vis[src] = true;

front = 0;

rear = 0;

q[rear++] = src;

while (front != rear)

{

int u = q[front++];

vis[u] = 0;

if (N == front)

front = 0;

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

{

int v = edge[i].v;

if (edge[i].c && edge[i].w + d[u] < d[v])

{

d[v] = edge[i].w + d[u];

if (!vis[v])

{

vis[v] = true;

q[rear++] = v;

if (N == rear)

rear = 0;

}

pv[v] = u;

pe[v] = i;

}

}

}

if (-1 == pv[sink])

break;

mxf = inf;

for (int i = sink; i != src; i = pv[i])

if (edge[pe[i]].c < mxf)

mxf = edge[pe[i]].c;

flow += mxf;

cost += d[sink] \* mxf;

for (int i = sink; i != src; i = pv[i])

{

edge[pe[i]].c -= mxf;

edge[pe[i] ^ 1].c += mxf;

}

}

return cost;

}

dinic最大流

/\*

\* 初始化:

\* ncount = 0;

\* memset(head, -1, sizeof(head));

\* addedge();加入所有边

\* 调用：dinic(s,t),s是源，t是汇

\* 返回：最大流

\*/

struct Edge

{

int next, v, f;

} edge[maxm \* 2];

int n, m;

int head[maxn];

int q[maxn];

bool vis[maxn];

int cur[maxn];

int dep[maxn];

int ncount;

int path[maxn];

void addedge(int a, int b, int f)

{

edge[ncount].v = b;

edge[ncount].f = f;

edge[ncount].next = head[a];

head[a] = ncount++;

}

void bfs(int s, int t)

{

memset(vis, 0, sizeof(vis));

memset(dep, -1, sizeof(dep));

int front = 0, rear = 0;

q[rear++] = s;

vis[s] = true;

dep[s] = 0;

while (front != rear && !vis[t])

{

int u = q[front++];

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

{

int v = edge[i].v;

if (!vis[v] && edge[i].f > 0)

{

q[rear++] = v;

vis[v] = true;

dep[v] = dep[u] + 1;

}

}

}

}

int dinic(int s, int t)

{

int ret = 0;

while (true)

{

bfs(s, t);

if (dep[t] == -1)

break;

int path\_n = 0;

int x = s;

memcpy(cur, head, sizeof(cur));

while (true)

{

if (x == t)

{

int mink = -1, delta = inf;

for (int i = 0; i < path\_n; ++i)

{

if (edge[path[i]].f < delta)

{

delta = edge[path[i]].f;

mink = i;

}

}

for (int i = 0; i < path\_n; ++i)

{

edge[path[i]].f -= delta;

edge[path[i] ^ 1].f += delta;

}

ret += delta;

path\_n = mink;

if (path\_n)

x = edge[path[path\_n - 1]].v;

else

x = s;

}

int e;

for (e = cur[x]; ~e; e = edge[e].next)

{

if (edge[e].f == 0)

continue;

int y = edge[e].v;

if (dep[x] + 1 == dep[y])

break;

}

cur[x] = e;

if (~e)

{

path[path\_n++] = e;

x = edge[e].v;

}

else

{

if (path\_n == 0)

break;

dep[x] = -1;

--path\_n;

if (path\_n)

x = edge[path[path\_n - 1]].v;

else

x = s;

}

}

}

return ret;

}

1. 图论

邻接表的存储方法

传统的存储方法，是用一个指针数组head[]，head中的每个元素作为链表头来引导一个edge类型的链表。如下图所示：



现在我们来介绍一下另一种数组模拟链表的存储邻接表的方法。在新方法中保持原方法的逻辑不变，只是把原方法开辟的分散的链表元素的空间，变为一段连续的数组空间。在这段数组空间中，包含了所有链表的所有元素，如下图所示：



既然是用数组存储，我们就可以用数组下标来代替链表指针。

本书中的大部分邻接表都采用了上述存储方式。

拓扑排序

/\*\*

\* 对一个有向无环图进行拓扑排序是指将图中的所有顶点

\* 排成一个序列， 该序列满足：若存在一条从i到j的边，

\* 则i在序列中排在j的前面。先调用buildmap函数建图

\* (具体根据题目要求修改)，然后调用toposort函数，

\* 函数的返回值为输出的顶点数，如果返回值不为N(顶

\* 点数)，则表示图中有环。排序复杂度为O(N + E)，注

\* 意排序结果可能不唯一，需要根据题目进行修改

\*/

const int MAXN = 2000;

int topo[MAXN], degree[MAXN], N;

bool graph[MAXN][MAXN];

void buildmap(int M) {

memset(graph, false, sizeof(graph));

memset(degree, 0, sizeof(degree));

int a, b;

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

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

if (!graph[a - 1][b - 1]) {

graph[a - 1][b - 1] = true;

degree[b - 1]++;

}

}

}

int toposort() {

stack<int> S;

int index = 0;

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

if(degree[i] == 0) {

S.push(i);

}

}

while(!S.empty()) {

int cur = S.top();

topo[index++] = cur;

S.pop();

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

if(graph[cur][i]) {

if(--degree[i] == 0) {

S.push(i);

}

}

}

}

return index;

}

强连通分支

/\*

\* 初始化： memset(head, -1, sizeof(head));

\* addedge()加入边

\* 调用：work()

\* 返回：cpn[]每个点所属的强连通分支号，cpn\_num强连通分支数

\*/

struct Edge

{

int v, next;

} edge[maxm];

int n, m, ncount, idx;

int head[maxn];

int dfn[maxn], low[maxn];

bool vis[maxn], instk[maxn];

int stk[maxn], top, cpn\_num, cpn[maxn];

void addedge(int a, int b)

{

//printf("%d %d\n", a, b);

edge[ncount].v = b;

edge[ncount].next = head[a];

head[a] = ncount++;

}

void tarjan(int u)

{

memset(vis, 0, sizeof(vis));

memset(instk, 0, sizeof(instk));

vis[u] = true;

stk[top++] = u;

instk[u] = true;

dfn[u] = low[u] = idx++;

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

{

int v = edge[i].v;

if (!vis[v])

{

tarjan(v);

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

}

else if (instk[v])

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

}

if (dfn[u] != low[u])

return;

int v;

do

{

v = stk[--top];

instk[v] = false;

cpn[v] = cpn\_num;

} while (u != v);

cpn\_num++;

}

int work()

{

idx = 0;

top = 0;

cpn\_num = 0;

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

if (!vis[i])

tarjan(i);

}

**无向图最小割**

/\*

\* 初始化：map邻接矩阵，memset(combine, 0, sizeof(combine));

\* 调用：stoer\_wagner()

\* 返回：最小割值

\*/

#define inf 0x3f3f3f3f

int n, m;

int map[maxn][maxn];

bool combine[maxn];

int s, e;

bool vis[maxn];

int w[maxn];

int mincut;

void search() {

memset(vis, 0, sizeof(vis));

memset(w, 0, sizeof(w));

while (true) {

int temp = -inf;

int x = -1;

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

if (!combine[i] && !vis[i] && w[i] > temp) {

temp = w[i];

x = i;

}

if (x == -1)

return;

s = e;

e = x;

mincut = temp;

vis[x] = true;

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

if (!combine[i] && !vis[i])

w[i] += map[x][i];

}

}

int stoer\_wagner() {

int ret = inf;

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

search();

ret = min(ret, mincut);

combine[e] = true;

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

if (!combine[j]) {

map[s][j] += map[e][j];

map[j][s] += map[j][e];

}

}

return ret;

}

spfa算法

/\*

\* 初始化：init()

\* 调用：spfa()

\* 返回：spfa()返回是否存在负权回路，dist[]为最短距离。

\* 注意：图不连通的情况下，一次spfa不能到达所有点。

\*/

struct Edge

{

int v, w, next;

} edge[maxm];

int n, m;

int dist[maxn];

bool vis[maxn];

int times[maxn];

int q[maxn];

int head[maxn];

int ecount;

void addedge(int a, int b, int w)

{

edge[ecount].v = b;

edge[ecount].w = w;

edge[ecount].next = head[a];

head[a] = ecount;

ecount++;

}

void init()

{

memset(head, -1, sizeof(head));

memset(vis, 0, sizeof(vis));

memset(times, 0, sizeof(times));

ecount = 0;

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

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

dist[i] = inf;

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

{

int a, b, x;

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

a--;

b--;

addedge(a, b, x);

addedge(b, a, x);

}

}

bool relax(int u, Edge &e)

{

if (dist[e.v] > dist[u] + e.w)

{

dist[e.v] = dist[u] + e.w;

return true;

}

return false;

}

bool spfa(int x)

{

int front = 0;

int rear = 1;

q[front] = x;

dist[x] = 0;

vis[x] = true;

while (front != rear)

{

int temp = q[front++];

if (front == maxn)

front = 0;

vis[temp] = false;

for (int i = head[temp]; i != -1; i = edge[i].next)

{

if (relax(temp, edge[i]) && !vis[edge[i].v])

{

q[rear++] = edge[i].v;

times[edge[i].v]++;

if (times[edge[i].v] > n)

return true;

if (rear == maxn)

rear = 0;

}

}

}

return false;

}

求割点

/\*

\* 调用：getpoint()

\* 返回：is[i]表示i点是否为割点

\*/

int n, dfn[maxn], low[maxn], cnt;

bool is[maxn];

bool g[maxn][maxn];

void dfs(int fa, int u)

{

low[u] = dfn[u] = cnt++;

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

if (g[u][i] && i != fa)

{

if (dfn[i] == -1)

{

dfs(u, i);

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

if (dfn[u] <= low[i])

is[u] = true;

;

}

else

low[u] = min(low[u], dfn[i]);

}

}

void getpoint()

{

memset(dfn, -1, sizeof(dfn));

memset(is, 0, sizeof(is));

cnt = 0;

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

if (dfn[i] == -1)

{

dfn[i] = cnt++;

int x = 0;

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

if (g[i][j] && dfn[j] == -1)

{

dfs(i, j);

x++;

}

if (x > 1)

is[i] = true;

}

}

求边双连通分支

/\*

\* 初始化：input()加边

\* 调用：tarjan(n)

\* 返回：加几条边可变为双连通图，low[i]表示点i所属的双双连通分支编号，tcount为双连通分支数

\*/

struct Edge

{

int v, next;

} edge[maxm];

int n, m;

int head[maxn];

bool hash[maxn][maxn];

int ecount, tcount;//tcount为双连通分支数

int dfn[maxn], vis[maxn], degree[maxn];

int low[maxn];//low[i]表示点i所属的双双连通分支编号

void addedge(int a, int b)

{

edge[ecount].v = b;

edge[ecount].next = head[a];

head[a] = ecount;

hash[a][b] = hash[b][a] = true;

ecount++;

}

void input()

{

memset(head, -1, sizeof(head));

ecount = 0;

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

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

{

int a, b;

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

a--;

b--;

if (hash[a][b])

continue;

addedge(a, b);

addedge(b, a);

}

}

void dfs(int fa, int u)

{

vis[u] = true;

low[u] = dfn[u] = tcount++;

for (int i = head[u]; i != -1; i = edge[i].next)

{

int v = edge[i].v;

if (v == fa)

continue;

if (!vis[v])

dfs(u, v);

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

}

}

int tarjan()

{

memset(dfn, 0, sizeof(dfn));

memset(vis, 0, sizeof(vis));

memset(degree, 0, sizeof(degree));

tcount = 0;

dfs(0, 0);

int ret = 0;

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

for (int j = head[i]; j != -1; j = edge[j].next)

{

int v = edge[j].v;

if (low[i] != low[v])

degree[low[i]]++;

}

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

if (degree[i] == 1)

ret++;

return (ret + 1) / 2;

}

bellman\_ford邻接阵形式

//单源最短路径,bellman\_ford算法,邻接阵形式,复杂度O(n^3)

//求出源s到所有点的最短路经,传入图的大小n和邻接阵mat

//返回到各点最短距离min[]和路径pre[],pre[i]记录s到i路径上i的父结点,pre[s]=-1

//可更改路权类型,路权可为负,若图包含负环则求解失败,返回0

//优化:先删去负边使用dijkstra求出上界,加速迭代过程

#define MAXN 200

#define inf 1000000000

typedef int elem\_t;

int bellman\_ford(int n,elem\_t mat[][MAXN],int s,elem\_t\* min,int\* pre){

int v[MAXN],i,j,k,tag;

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

min[i]=inf,v[i]=0,pre[i]=-1;

for (min[s]=0,j=0;j<n;j++){

for (k=-1,i=0;i<n;i++)

if (!v[i]&&(k==-1||min[i]<min[k]))

k=i;

for (v[k]=1,i=0;i<n;i++)

if !v[i]&&mat[k][i]>=0&&min[k]+mat[k][i]<min[i])

min[i]=min[k]+mat[pre[i]=k][i];

}

for (tag=1,j=0;tag&&j<=n;j++)

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

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

if (min[k]+mat[k][i]<min[i])

min[i]=min[k]+mat[pre[i]=k][i],tag=1;

return j<=n;

}

bellman-ford邻接表形式

/\*

\* 初始化：input()

\* 调用：bellman(起点)

\* 返回：dist[]到各点最短距离

\*/

struct Edge

{

int u, v, w;

} edge[maxm];

int n, m;

int dist[maxn];

int ecount;

int s, e;

void addedge(int a, int b, int w)

{

edge[ecount].u = a;

edge[ecount].v = b;

edge[ecount].w = w;

ecount++;

}

void input()

{

ecount = 0;

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

{

int a, b, w;

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

addedge(a, b, w);

addedge(b, a, w);

}

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

}

bool relax(int u, int v, int w)

{

if (dist[v] > dist[u] + w)

{

dist[v] = dist[u] + w;

return true;

}

return false;

}

bool bellman(int s)

{

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

dist[i] = inf;

dist[s] = 0;

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

{

bool flag = false;

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

if (relax(edge[j].u, edge[j].v, edge[j].w))

flag = true;

if (!flag)

break;

}

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

if (relax(edge[j].u, edge[j].v, edge[j].w))

return false;

return true;

}

dijkstra邻接表

/\*

\* 初始化：input()

\* 调用：dijkstra(起点)

\* 返回：dist[]到各点最短距离

\*/

struct Edge

{

int v, w, next;

} edge[maxm];

int n, m;

int dist[maxn];

bool vis[maxn];

int head[maxn];

int ecount;

int s, e;

void addedge(int a, int b, int w)

{

edge[ecount].v = b;

edge[ecount].w = w;

edge[ecount].next = head[a];

head[a] = ecount;

ecount++;

}

void input()

{

memset(head, -1, sizeof(head));

ecount = 0;

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

{

int a, b, w;

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

addedge(a, b, w);

addedge(b, a, w);

}

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

}

void dijkstra(int s)

{

memset(vis, 0, sizeof(vis));

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

dist[i] = inf;

dist[s] = 0;

while (1)

{

int u = -1;

int min\_dist = inf;

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

if (!vis[i] && dist[i] < min\_dist)

{

u = i;

min\_dist = dist[i];

}

if (u == -1)

return;

vis[u] = true;

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

{

int v = edge[i].v;

dist[v] = min(dist[v], dist[u] + edge[i].w);

}

}

}

djkstra邻接表+堆优化

/\*

\* 初始化：input()

\* 调用：dijkstra(起点)

\* 返回：dist[]到各点最短距离

\* 注意：堆优化适用于稀疏图

\*/

struct Edge

{

int v, w, next;

} edge[maxm];

struct Elem

{

int dis, id;

Elem()

{}

Elem(int dd, int ii):dis(dd), id(ii)

{}

};

int n, m;

int dist[maxn];

bool vis[maxn];

int head[maxn];

int ecount;

int s, e;

bool operator < (const Elem &a, const Elem &b)

{

return a.dis > b.dis;

}

void addedge(int a, int b, int w)

{

edge[ecount].v = b;

edge[ecount].w = w;

edge[ecount].next = head[a];

head[a] = ecount;

ecount++;

}

void input()

{

memset(head, -1, sizeof(head));

ecount = 0;

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

{

int a, b, w;

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

addedge(a, b, w);

addedge(b, a, w);

}

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

}

void dijkstra(int s)

{

memset(vis, 0, sizeof(vis));

priority\_queue <Elem> pq;

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

dist[i] = inf;

dist[s] = 0;

pq.push(Elem(0, s));

while (!pq.empty())

{

Elem a;

do{

a = pq.top();

pq.pop();

}while(vis[a.id] && !pq.empty());

int u = a.id;

vis[u] = true;

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

{

int v = edge[i].v;

if (dist[v] > dist[u] + edge[i].w)

{

dist[v] = dist[u] + edge[i].w;

pq.push(Elem(dist[v], v));

}

}

}

}

dijkstra邻接阵形式?

/\*\*

\* 单源最短路dijkstra算法,邻接阵形式,复杂度O(n^2).

\* graph存图的邻接阵(graph[i][j]=INF表示i与j无直

\* 接相连的边.建好图以后,每一次调用dijistra(s,e)

\* 就可返回s到e的最短路径长度.N为顶点数.可更改边

\* 权类型，但必须非负。无向图有向图均适用。

\*/

int N;//图的顶点数

const int SIZE = 10;

typedef int typec;

const typec INF = 0x7fffffff;

typec graph[SIZE][SIZE];

typec dijistra(int s, int e) {

int i, j, k;

typec mind, minf, D[SIZE];

bool visited[SIZE];

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

visited[i] = false;

D[i] = graph[s][i];

}

visited[s] = 1;

D[s] = 0;

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

mind = INF;

minf = INF;

k = 0;

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

if (visited[j]) {

continue;

}

if (D[j] < mind) {

k = j;

mind = D[j];

}

}

visited[k] = true;

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

if (!visited[j]) {

if (D[k] < D[j] - graph[k][j]) {

D[j] = D[k] + graph[k][j];

}

}

}

}

return D[e];

}

floyd邻接矩阵形式

/\*\*

\* 多源最短路Floyd算法,复杂度O(n^3).其实核心就三个循环

\* 代码中graph存图的邻接阵(graph[i][j]=INF表示i与j无直

\* 接相连的边.path[i][j]存的是从i到j的路径中i的下一个

\* 节点的节点号(从0开始).N表示图的规模(即顶点数)，

\* SIZE为规模上限。

\*/

typedef int typec;

const int SIZE = 10;

const typec INF = 0x7fffffff;

typec graph[SIZE][SIZE];

int path[SIZE][SIZE], N;

void Floyd() {

int i, j, k;

for (k = 0; k < N; k++) {

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

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

if(graph[i][k] < graph[i][j] - graph[k][j]) {

graph[i][j] = graph[i][k] + graph[k][j];

path[i][j] = path[i][k];

}

}

}

}

}

普里姆（加点法）

/\*

\* 初始化：cost[][]邻接矩阵

\* 返回：最小生成树的总长度

\*/

double prim(int n, double cost[][maxn])

{

double lowcost[maxn];

bool vis[maxn];

memset(vis, 0, sizeof(vis));

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

lowcost[i] = -1;

lowcost[0] = 0;

vis[0] = true;

int pre = 0;

double ans = 0;

while (1)

{

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

if (lowcost[i] > lowcost[pre] + cost[i][pre] || lowcost[i] == -1)

lowcost[i] = lowcost[pre] + cost[i][pre];

int best = 1000000000;;

pre = -1;

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

if (!vis[i] && lowcost[i] < best)

{

best = lowcost[i];

pre = i;

}

if (pre == -1)

break;

ans += lowcost[pre];

vis[pre] = true;

lowcost[pre] = 0;

}

return ans;

}

克鲁斯卡尔（加边法）

const int MAXM = 200010;

const int MAXN = 200010;

typedef int typec;

typedef struct {

int s, e; typec len;

} MyEdge;

int myset[MAXM], myheight[MAXM];

MyEdge edges[MAXN];

int N, M;

inline bool operator<(const MyEdge &e1, const MyEdge &e2) {

return e1.len < e2.len;

}

void initset() {

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

myset[i] = i; myheight[i] = 1;

}

}

int myfind(int x) {

while(myset[x] != x) { x = myset[x]; }

return x;

}

void mymerge(int a, int b) {

if(myheight[a] == myheight[b]) {

myheight[a]++;

myset[b] = a;

}else if(myheight[a] < myheight[b]) {

myset[a] = b;

}else {

myset[b] = a;

}

}

int main() {

int ans, x, y, z;

while(scanf("%d%d", &M, &N) == 2) {

ans = 0;

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

scanf("%d%d%d", &x, &y, &z);

edges[i].s = x;

edges[i].e = y;

edges[i].len = z;

}

sort(edges, edges + N); initset();

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

x = edges[i].s;

y = edges[i].e;

z = edges[i].len;

if(myfind(x) == myfind(y)) { continue; }

mymerge(myfind(x), myfind(y));

ans += z;

}

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

}

return 0;

}

2-sat问题

/\*

\* 初始化：

\* ecount = 0;

\* memset(head, -1, sizeof(head));

\* enemy(a,b)加入所有敌对关系

\* 调用：work()

\* 返回：是否有可行解

\*/

int head[maxn], ecount;

int id[maxn], dfn[maxn], low[maxn];

bool vis[maxn];

int stk[maxn], top;

int cnt1, cnt0;

void addedge(int a, int b)

{

edge[ecount].v = b;

edge[ecount].next = head[a];

head[a] = ecount++;

}

void enemy(int a, int b)

{

addedge(a, (b ^ 1));

addedge(b, (a ^ 1));

}

void dfs(int u)

{

low[u] = dfn[u] = cnt0++;

stk[top++] = u;

vis[u] = true;

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

{

int v = edge[i].v;

if (!~dfn[v])

{

dfs(v);

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

continue;

}

if (vis[v])

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

}

if (dfn[u] != low[u])

return;

do

{

top--;

vis[stk[top]] = false;

id[stk[top]] = cnt1;

}while (stk[top] != u);

cnt1++;

}

bool work()

{

memset(id, -1, sizeof(id));

memset(dfn, -1, sizeof(dfn));

cnt0 = cnt1 = top = 0;

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

if (!~dfn[i])

dfs(i);

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

if (id[i] == id[i + 1])

return false;

return true;

}

1. 计算几何

几何题注意事项

1. 注意舍入方式(0.5的舍入方向);防止输出-0.

2. 几何题注意多测试不对称数据.

3. 整数几何注意xmult和dmult是否会出界;浮点几何注意eps的使用.

4. 避免使用斜率;注意除数是否会为0.

5. 公式一定要化简后再代入.

6. 判断同一个2\*PI域内两角度差应该是

abs(a1-a2)<beta||abs(a1-a2)>pi+pi-beta;

相等应该是

abs(a1-a2)<eps||abs(a1-a2)>pi+pi-eps;

7. 需要的话尽量使用atan2,注意:atan2(0,0)=0,

atan2(1,0)=pi/2,atan2(-1,0)=-pi/2,atan2(0,1)=0,atan2(0,-1)=pi.

8. cross product = |u|\*|v|\*sin(a)

dot product = |u|\*|v|\*cos(a)

9. (P1-P0)x(P2-P0)结果的意义:

正: <P0,P1>在<P0,P2>顺时针(0,pi)内

负: <P0,P1>在<P0,P2>逆时针(0,pi)内

0 : <P0,P1>,<P0,P2>共线,夹角为0或pi

10. 误差限缺省使用1e-8!

常用几何公式

三角形

1. 半周长 P=(a+b+c)/2

2. 面积 S=aHa/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

几何题公共头

const int MAXN = 1000;

const int offset = 10000;

const double eps = 1e-8;

#define zero(x) (((x)>0?(x):-(x))<eps)

#define \_sign(x) ((x)>eps?1:((x)<-eps?2:0))

typedef int typec;

typedef struct {

typec x;

typec y;

} MyPoint;

typedef struct{

MyPoint a, b;

}MyLine;

inline double xmult(MyPoint p1, MyPoint p2, MyPoint p0) {

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

}

叉积

(P1-P0)x(P2-P0)

Inline typec xmult(MyPoint p1,MyPoint p2,MyPoint p0) {

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

}

inline typec xmult(typec x1, typec y1, typec x2, typec y2, typec x0, typec y0) {

return (x1 - x0) \* (y2 - y0) - (x2 - x0) \* (y1 - y0);

}

点积

(P1-P0).(P2-P0)

typec dmult(MyPoint p1,MyPoint p2,MyPoint p0) {

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

}

typec dmult(typec x1, typec y1, typec x2, typec y2, typec x0, typec y0) {

return (x1 - x0) \* (x2 - x0) + (y1 - y0) \* (y2 - y0);

}

多边形

多边形凹凸判定

//顶点按顺(逆)时针给出,允许相邻边共线

bool is\_convex(int n, MyPoint\* p) {

int i, s[3] = { 1, 1, 1 };

for (i = 0; i < n && (s[1] | s[2]); i++) {

s[\_sign(xmult(p[(i + 1) % n], p[(i + 2) % n], p[i]))] = 0;

}

return s[1] | s[2];

}

//顶点按顺(逆)时针给出,不许相邻边共线

int is\_convex\_v2(int n,point\* p){

int i,s[3]={1,1,1};

for (i=0;i<n&&s[0]&&s[1]|s[2];i++)

s[\_sign(xmult(p[(i+1)%n],p[(i+2)%n],p[i]))]=0;

return s[0]&&s[1]|s[2];

}

判点在凸多边形内或多边形边上

//顶点按顺时针或逆时针给出

int inside\_convex(point q,int n,point\* p){

int i,s[3]={1,1,1};

for (i=0;i<n&&s[1]|s[2];i++)

s[\_sign(xmult(p[(i+1)%n],q,p[i]))]=0;

return s[1]|s[2];

}

//顶点按顺时针或逆时针给出,在多边形边上返回0

int inside\_convex\_v2(point q,int n,point\* p){

int i,s[3]={1,1,1};

for (i=0;i<n&&s[0]&&s[1]|s[2];i++)

s[\_sign(xmult(p[(i+1)%n],q,p[i]))]=0;

return s[0]&&s[1]|s[2];

}

//判点在任意多边形内,顶点按顺时针或逆时针给出

//on\_edge表示点在多边形边上时的返回值,offset为多边形坐标上限

int inside\_polygon(point q,int n,point\* p,int on\_edge=1){

point q2;

int i=0,count;

while (i<n)

for (count=i=0,q2.x=rand()+offset,q2.y=rand()+offset;i<n;i++)

if (zero(xmult(q,p[i],p[(i+1)%n]))&&(p[i].x-q.x)\*(p[(i+1)%n].x-q.x)<eps&&(p[i].y-q.y)\*(p[(i+1)%n].y-q.y)<eps)

return on\_edge;

else if (zero(xmult(q,q2,p[i])))

break;

else if (xmult(q,p[i],q2)\*xmult(q,p[(i+1)%n],q2)<-eps&&xmult(p[i],q,p[(i+1)%n])\*xmult(p[i],q2,p[(i+1)%n])<-eps)

count++;

return count&1;

}

inline int opposite\_side(point p1,point p2,point l1,point l2){

return xmult(l1,p1,l2)\*xmult(l1,p2,l2)<-eps;

}

inline int dot\_online\_in(point p,point l1,point l2){

return zero(xmult(p,l1,l2))&&(l1.x-p.x)\*(l2.x-p.x)<eps&&(l1.y-p.y)\*(l2.y-p.y)<eps;

}

//判线段在任意多边形内,顶点按顺时针或逆时针给出,与边界相交返回1

int inside\_polygon(point l1,point l2,int n,point\* p){

point t[MAXN],tt;

int i,j,k=0;

if (!inside\_polygon(l1,n,p)||!inside\_polygon(l2,n,p))

return 0;

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

if (opposite\_side(l1,l2,p[i],p[(i+1)%n])&&opposite\_side(p[i],p[(i+1)%n],l1,l2))

return 0;

else if (dot\_online\_in(l1,p[i],p[(i+1)%n]))

t[k++]=l1;

else if (dot\_online\_in(l2,p[i],p[(i+1)%n]))

t[k++]=l2;

else if (dot\_online\_in(p[i],l1,l2))

t[k++]=p[i];

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

for (j=i+1;j<k;j++){

tt.x=(t[i].x+t[j].x)/2;

tt.y=(t[i].y+t[j].y)/2;

if (!inside\_polygon(tt,n,p))

return 0;

}

return 1;

}

point intersection(line u,line v){

point ret=u.a;

double t=((u.a.x-v.a.x)\*(v.a.y-v.b.y)-(u.a.y-v.a.y)\*(v.a.x-v.b.x))

/((u.a.x-u.b.x)\*(v.a.y-v.b.y)-(u.a.y-u.b.y)\*(v.a.x-v.b.x));

ret.x+=(u.b.x-u.a.x)\*t;

ret.y+=(u.b.y-u.a.y)\*t;

return ret;

}

point barycenter(point a,point b,point c){

line u,v;

u.a.x=(a.x+b.x)/2;

u.a.y=(a.y+b.y)/2;

u.b=c;

v.a.x=(a.x+c.x)/2;

v.a.y=(a.y+c.y)/2;

v.b=b;

return intersection(u,v);

}

多边形重心

point barycenter(int n,point\* p){

point ret,t;

double t1=0,t2;

int i;

ret.x=ret.y=0;

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

if (fabs(t2=xmult(p[0],p[i],p[i+1]))>eps){

t=barycenter(p[0],p[i],p[i+1]);

ret.x+=t.x\*t2;

ret.y+=t.y\*t2;

t1+=t2;

}

if (fabs(t1)>eps)

ret.x/=t1,ret.y/=t1;

return ret;

}

多边形切割

//多边形切割

//可用于半平面交

#define MAXN 100

#define eps 1e-8

#define zero(x) (((x)>0?(x):-(x))<eps)

struct point{double x,y;};

double xmult(point p1,point p2,point p0){

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

}

int same\_side(point p1,point p2,point l1,point l2){

return xmult(l1,p1,l2)\*xmult(l1,p2,l2)>eps;

}

point intersection(point u1,point u2,point v1,point v2){

point ret=u1;

double t=((u1.x-v1.x)\*(v1.y-v2.y)-(u1.y-v1.y)\*(v1.x-v2.x))

/((u1.x-u2.x)\*(v1.y-v2.y)-(u1.y-u2.y)\*(v1.x-v2.x));

ret.x+=(u2.x-u1.x)\*t;

ret.y+=(u2.y-u1.y)\*t;

return ret;

}

//将多边形沿l1,l2确定的直线切割在side侧切割,保证l1,l2,side不共线

void polygon\_cut(int& n,point\* p,point l1,point l2,point side){

point pp[100];

int m=0,i;

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

if (same\_side(p[i],side,l1,l2))

pp[m++]=p[i];

if (!same\_side(p[i],p[(i+1)%n],l1,l2)&&!(zero(xmult(p[i],l1,l2))&&zero(xmult(p[(i+1)%n],l1,l2))))

pp[m++]=intersection(p[i],p[(i+1)%n],l1,l2);

}

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

if (!i||!zero(pp[i].x-pp[i-1].x)||!zero(pp[i].y-pp[i-1].y))

p[n++]=pp[i];

if (zero(p[n-1].x-p[0].x)&&zero(p[n-1].y-p[0].y))

n--;

if (n<3)

n=0;

}

浮点函数

两点距离

double mydistance(const MyPoint &p1, const MyPoint &p2) {

return sqrt((p1.x - p2.x) \* (p1.x - p2.x) + (p1.y - p2.y) \* (p1.y - p2.y));

}

double distance(double x1,double y1,double x2,double y2){

return sqrt((x1-x2)\*(x1-x2)+(y1-y2)\*(y1-y2));

}

判三点共线

int dots\_inline(point p1,point p2,point p3){

return zero(xmult(p1,p2,p3));

}

int dots\_inline(double x1,double y1,double x2,double y2,double x3,double y3){

return zero(xmult(x1,y1,x2,y2,x3,y3));

}

判断点和线段关系

inline int dot\_online\_in(const MyPoint &p, const MyLine &l) {

return dot\_online\_in(p.x, p.y, l.a.x, l.a.y, l.b.x, l.b.y);

}

/\*

\* 判断点和线段关系

\* 返回

\* 0表示在线段上

\* 1表示点在线段延长线上

\* 2表示点在线段端点上

\* -1表示点与线段不共线

\*/

inline int dot\_online\_in(typec x, typec y, typec x1, typec y1, typec x2,

typec y2) {

if ((zero(x-x1) && zero(y-y1)) || (zero(x-x2) && zero(y-y2))) {

return 2;

}

if (!zero(xmult(x,y,x1,y1,x2,y2))) {

return -1;

}

if ((x1 - x) \* (x2 - x) < eps && (y1 - y) \* (y2 - y) < eps) {

return 0;

}

return 1;

}

判两点与线段位置关系

//判两点在线段同侧,点在线段上返回0

int same\_side(point p1,point p2,line l){

return xmult(l.a,p1,l.b)\*xmult(l.a,p2,l.b)>eps;

}

int same\_side(point p1,point p2,point l1,point l2){

return xmult(l1,p1,l2)\*xmult(l1,p2,l2)>eps;

}

//判两点在线段异侧,点在线段上返回0

int opposite\_side(point p1,point p2,line l){

return xmult(l.a,p1,l.b)\*xmult(l.a,p2,l.b)<-eps;

}

int opposite\_side(point p1,point p2,point l1,point l2){

return xmult(l1,p1,l2)\*xmult(l1,p2,l2)<-eps;

}

判两直线平行

int parallel(line u,line v){

return zero((u.a.x-u.b.x)\*(v.a.y-v.b.y)-(v.a.x-v.b.x)\*(u.a.y-u.b.y));

}

int parallel(point u1,point u2,point v1,point v2){

return zero((u1.x-u2.x)\*(v1.y-v2.y)-(v1.x-v2.x)\*(u1.y-u2.y));

}

判两直线垂直

int perpendicular(line u,line v){

return zero((u.a.x-u.b.x)\*(v.a.x-v.b.x)+(u.a.y-u.b.y)\*(v.a.y-v.b.y));

}

int perpendicular(point u1,point u2,point v1,point v2){

return zero((u1.x-u2.x)\*(v1.x-v2.x)+(u1.y-u2.y)\*(v1.y-v2.y));

}

判两线段相交

//包括端点和部分重合

int intersect\_in(line u,line v){

if (!dots\_inline(u.a,u.b,v.a)||!dots\_inline(u.a,u.b,v.b))

return !same\_side(u.a,u.b,v)&&!same\_side(v.a,v.b,u);

return dot\_online\_in(u.a,v)||dot\_online\_in(u.b,v)||dot\_online\_in(v.a,u)||dot\_online\_in(v.b,u);

}

int intersect\_in(point u1,point u2,point v1,point v2){

if (!dots\_inline(u1,u2,v1)||!dots\_inline(u1,u2,v2))

return !same\_side(u1,u2,v1,v2)&&!same\_side(v1,v2,u1,u2);

return dot\_online\_in(u1,v1,v2)||dot\_online\_in(u2,v1,v2)||dot\_online\_in(v1,u1,u2)||dot\_online\_in(v2,u1,u2);

}

//不包括端点和部分重合

int intersect\_ex(line u,line v){

return opposite\_side(u.a,u.b,v)&&opposite\_side(v.a,v.b,u);

}

int intersect\_ex(point u1,point u2,point v1,point v2){

return opposite\_side(u1,u2,v1,v2)&&opposite\_side(v1,v2,u1,u2);

}

计算两直线交点

//注意事先判断直线是否平行!

//线段交点请另外判线段相交(同时还是要判断是否平行!)

point intersection(line u,line v){

point ret=u.a;

double t=((u.a.x-v.a.x)\*(v.a.y-v.b.y)-(u.a.y-v.a.y)\*(v.a.x-v.b.x))

/((u.a.x-u.b.x)\*(v.a.y-v.b.y)-(u.a.y-u.b.y)\*(v.a.x-v.b.x));

ret.x+=(u.b.x-u.a.x)\*t;

ret.y+=(u.b.y-u.a.y)\*t;

return ret;

}

point intersection(point u1,point u2,point v1,point v2){

point ret=u1;

double t=((u1.x-v1.x)\*(v1.y-v2.y)-(u1.y-v1.y)\*(v1.x-v2.x))

/((u1.x-u2.x)\*(v1.y-v2.y)-(u1.y-u2.y)\*(v1.x-v2.x));

ret.x+=(u2.x-u1.x)\*t;

ret.y+=(u2.y-u1.y)\*t;

return ret;

}

点到直线上的最近点

point ptoline(point p,line l){

point t=p;

t.x+=l.a.y-l.b.y,t.y+=l.b.x-l.a.x;

return intersection(p,t,l.a,l.b);

}

point ptoline(point p,point l1,point l2){

point t=p;

t.x+=l1.y-l2.y,t.y+=l2.x-l1.x;

return intersection(p,t,l1,l2);

}

点到直线距离

double disptoline(point p,line l){

return fabs(xmult(p,l.a,l.b))/distance(l.a,l.b);

}

double disptoline(point p,point l1,point l2){

return fabs(xmult(p,l1,l2))/distance(l1,l2);

}

double disptoline(double x,double y,double x1,double y1,double x2,double y2){

return fabs(xmult(x,y,x1,y1,x2,y2))/distance(x1,y1,x2,y2);

}

点到线段上的最近点

point ptoseg(point p,line l){

point t=p;

t.x+=l.a.y-l.b.y,t.y+=l.b.x-l.a.x;

if (xmult(l.a,t,p)\*xmult(l.b,t,p)>eps)

return distance(p,l.a)<distance(p,l.b)?l.a:l.b;

return intersection(p,t,l.a,l.b);

}

point ptoseg(point p,point l1,point l2){

point t=p;

t.x+=l1.y-l2.y,t.y+=l2.x-l1.x;

if (xmult(l1,t,p)\*xmult(l2,t,p)>eps)

return distance(p,l1)<distance(p,l2)?l1:l2;

return intersection(p,t,l1,l2);

}

点到线段距离

double disptoseg(point p,line l){

point t=p;

t.x+=l.a.y-l.b.y,t.y+=l.b.x-l.a.x;

if (xmult(l.a,t,p)\*xmult(l.b,t,p)>eps)

return distance(p,l.a)<distance(p,l.b)?distance(p,l.a):distance(p,l.b);

return fabs(xmult(p,l.a,l.b))/distance(l.a,l.b);

}

double disptoseg(point p,point l1,point l2){

point t=p;

t.x+=l1.y-l2.y,t.y+=l2.x-l1.x;

if (xmult(l1,t,p)\*xmult(l2,t,p)>eps)

return distance(p,l1)<distance(p,l2)?distance(p,l1):distance(p,l2);

return fabs(xmult(p,l1,l2))/distance(l1,l2);

}

矢量V以P为顶点逆时针旋转angle并放大scale倍

point rotate(point v,point p,double angle,double scale){

point ret=p;

v.x-=p.x,v.y-=p.y;

p.x=scale\*cos(angle);

p.y=scale\*sin(angle);

ret.x+=v.x\*p.x-v.y\*p.y;

ret.y+=v.x\*p.y+v.y\*p.x;

return ret;

}

面积

公共头

struct point{double x,y;};

//计算cross product (P1-P0)x(P2-P0)

double xmult(point p1,point p2,point p0){

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

}

double xmult(double x1,double y1,double x2,double y2,double x0,double y0){

return (x1-x0)\*(y2-y0)-(x2-x0)\*(y1-y0);

}

计算三角形面积

//输入三顶点

double area\_triangle(point p1,point p2,point p3){

return fabs(xmult(p1,p2,p3))/2;

}

double area\_triangle(double x1,double y1,double x2,double y2,double x3,double y3){

return fabs(xmult(x1,y1,x2,y2,x3,y3))/2;

}

//输入三边长

double area\_triangle(double a,double b,double c){

double s=(a+b+c)/2;

return sqrt(s\*(s-a)\*(s-b)\*(s-c));

}

计算多边形面积

//顶点按顺时针或逆时针给出

double area\_polygon(int n,point\* p){

double s1=0,s2=0;

int i;

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

s1+=p[(i+1)%n].y\*p[i].x,s2+=p[(i+1)%n].y\*p[(i+2)%n].x;

return fabs(s1-s2)/2;

}

球面

公共头

const double pi=acos(-1);

//计算圆心角lat表示纬度,-90<=w<=90,lng表示经度

//返回两点所在大圆劣弧对应圆心角,0<=angle<=pi

double angle(double lng1,double lat1,double lng2,double lat2){

double dlng=fabs(lng1-lng2)\*pi/180;

while (dlng>=pi+pi)

dlng-=pi+pi;

if (dlng>pi)

dlng=pi+pi-dlng;

lat1\*=pi/180,lat2\*=pi/180;

return acos(cos(lat1)\*cos(lat2)\*cos(dlng)+sin(lat1)\*sin(lat2));

}

计算距离,r为球半径

double line\_dist(double r,double lng1,double lat1,double lng2,double lat2){

double dlng=fabs(lng1-lng2)\*pi/180;

while (dlng>=pi+pi)

dlng-=pi+pi;

if (dlng>pi)

dlng=pi+pi-dlng;

lat1\*=pi/180,lat2\*=pi/180;

return r\*sqrt(2-2\*(cos(lat1)\*cos(lat2)\*cos(dlng)+sin(lat1)\*sin(lat2)));

}

计算球面距离,r为球半径

inline double sphere\_dist(double r,double lng1,double lat1,double lng2,double lat2){

return r\*angle(lng1,lat1,lng2,lat2);

}

三角形

公共头

struct point{double x,y;};

struct line{point a,b;};

double distance(point p1,point p2){

return sqrt((p1.x-p2.x)\*(p1.x-p2.x)+(p1.y-p2.y)\*(p1.y-p2.y));

}

point intersection(line u,line v){

point ret=u.a;

double t=((u.a.x-v.a.x)\*(v.a.y-v.b.y)-(u.a.y-v.a.y)\*(v.a.x-v.b.x))

/((u.a.x-u.b.x)\*(v.a.y-v.b.y)-(u.a.y-u.b.y)\*(v.a.x-v.b.x));

ret.x+=(u.b.x-u.a.x)\*t;

ret.y+=(u.b.y-u.a.y)\*t;

return ret;

}

外心

point circumcenter(point a,point b,point c){

line u,v;

u.a.x=(a.x+b.x)/2;

u.a.y=(a.y+b.y)/2;

u.b.x=u.a.x-a.y+b.y;

u.b.y=u.a.y+a.x-b.x;

v.a.x=(a.x+c.x)/2;

v.a.y=(a.y+c.y)/2;

v.b.x=v.a.x-a.y+c.y;

v.b.y=v.a.y+a.x-c.x;

return intersection(u,v);

}

内心

point incenter(point a,point b,point c){

line u,v;

double m,n;

u.a=a;

m=atan2(b.y-a.y,b.x-a.x);

n=atan2(c.y-a.y,c.x-a.x);

u.b.x=u.a.x+cos((m+n)/2);

u.b.y=u.a.y+sin((m+n)/2);

v.a=b;

m=atan2(a.y-b.y,a.x-b.x);

n=atan2(c.y-b.y,c.x-b.x);

v.b.x=v.a.x+cos((m+n)/2);

v.b.y=v.a.y+sin((m+n)/2);

return intersection(u,v);

}

垂心

point perpencenter(point a,point b,point c){

line u,v;

u.a=c;

u.b.x=u.a.x-a.y+b.y;

u.b.y=u.a.y+a.x-b.x;

v.a=b;

v.b.x=v.a.x-a.y+c.y;

v.b.y=v.a.y+a.x-c.x;

return intersection(u,v);

}

重心

//到三角形三顶点距离的平方和最小的点

//三角形内到三边距离之积最大的点

point barycenter(point a,point b,point c){

line u,v;

u.a.x=(a.x+b.x)/2;

u.a.y=(a.y+b.y)/2;

u.b=c;

v.a.x=(a.x+c.x)/2;

v.a.y=(a.y+c.y)/2;

v.b=b;

return intersection(u,v);

}

费马点

//到三角形三顶点距离之和最小的点

point fermentpoint(point a,point b,point c){

point u,v;

double step=fabs(a.x)+fabs(a.y)+fabs(b.x)+fabs(b.y)+fabs(c.x)+fabs(c.y);

int i,j,k;

u.x=(a.x+b.x+c.x)/3;

u.y=(a.y+b.y+c.y)/3;

while (step>1e-10)

for (k=0;k<10;step/=2,k++)

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

for (j=-1;j<=1;j++){

v.x=u.x+step\*i;

v.y=u.y+step\*j;

if (distance(u,a)+distance(u,b)+distance(u,c)>distance(v,a)+distance(v,b)+distance(v,c))

u=v;

}

return u;

}

三维几何

公共头

#define eps 1e-8

#define zero(x) (((x)>0?(x):-(x))<eps)

struct point3{double x,y,z;};

struct line3{point3 a,b;};

struct plane3{point3 a,b,c;};

三维叉积

point3 xmult(point3 u,point3 v){

point3 ret;

ret.x=u.y\*v.z-v.y\*u.z;

ret.y=u.z\*v.x-u.x\*v.z;

ret.z=u.x\*v.y-u.y\*v.x;

return ret;

}

三维点积

double dmult(point3 u,point3 v){

return u.x\*v.x+u.y\*v.y+u.z\*v.z;

}

矢量差 U - V

point3 subt(point3 u,point3 v){

point3 ret;

ret.x=u.x-v.x;

ret.y=u.y-v.y;

ret.z=u.z-v.z;

return ret;

}

取平面法向量

point3 pvec(plane3 s){

return xmult(subt(s.a,s.b),subt(s.b,s.c));

}

point3 pvec(point3 s1,point3 s2,point3 s3){

return xmult(subt(s1,s2),subt(s2,s3));

}

两点距离,单参数取向量大小

double distance(point3 p1,point3 p2){

return sqrt((p1.x-p2.x)\*(p1.x-p2.x)+(p1.y-p2.y)\*(p1.y-p2.y)+(p1.z-p2.z)\*(p1.z-p2.z));

}

向量大小

double vlen(point3 p){

return sqrt(p.x\*p.x+p.y\*p.y+p.z\*p.z);

}

判三点共线

int dots\_inline(point3 p1,point3 p2,point3 p3){

return vlen(xmult(subt(p1,p2),subt(p2,p3)))<eps;

}

判四点共面

int dots\_onplane(point3 a,point3 b,point3 c,point3 d){

return zero(dmult(pvec(a,b,c),subt(d,a)));

}

判点是否在线段上

//包括端点和共线

int dot\_online\_in(point3 p,line3 l){

return zero(vlen(xmult(subt(p,l.a),subt(p,l.b))))&&(l.a.x-p.x)\*(l.b.x-p.x)<eps&&

(l.a.y-p.y)\*(l.b.y-p.y)<eps&&(l.a.z-p.z)\*(l.b.z-p.z)<eps;

}

int dot\_online\_in(point3 p,point3 l1,point3 l2){

return zero(vlen(xmult(subt(p,l1),subt(p,l2))))&&(l1.x-p.x)\*(l2.x-p.x)<eps&&

(l1.y-p.y)\*(l2.y-p.y)<eps&&(l1.z-p.z)\*(l2.z-p.z)<eps;

}

//不包括端点

int dot\_online\_ex(point3 p,line3 l){

return dot\_online\_in(p,l)&&(!zero(p.x-l.a.x)||!zero(p.y-l.a.y)||!zero(p.z-l.a.z))&&

(!zero(p.x-l.b.x)||!zero(p.y-l.b.y)||!zero(p.z-l.b.z));

}

int dot\_online\_ex(point3 p,point3 l1,point3 l2){

return dot\_online\_in(p,l1,l2)&&(!zero(p.x-l1.x)||!zero(p.y-l1.y)||!zero(p.z-l1.z))&&

(!zero(p.x-l2.x)||!zero(p.y-l2.y)||!zero(p.z-l2.z));

}

判点是否在空间三角形上

//包括边界,三点共线无意义

int dot\_inplane\_in(point3 p,plane3 s){

return zero(vlen(xmult(subt(s.a,s.b),subt(s.a,s.c)))-vlen(xmult(subt(p,s.a),subt(p,s.b)))-

vlen(xmult(subt(p,s.b),subt(p,s.c)))-vlen(xmult(subt(p,s.c),subt(p,s.a))));

}

int dot\_inplane\_in(point3 p,point3 s1,point3 s2,point3 s3){

return zero(vlen(xmult(subt(s1,s2),subt(s1,s3)))-vlen(xmult(subt(p,s1),subt(p,s2)))-

vlen(xmult(subt(p,s2),subt(p,s3)))-vlen(xmult(subt(p,s3),subt(p,s1))));

}

//不包括边界,三点共线无意义

int dot\_inplane\_ex(point3 p,plane3 s){

return dot\_inplane\_in(p,s)&&vlen(xmult(subt(p,s.a),subt(p,s.b)))>eps&&

vlen(xmult(subt(p,s.b),subt(p,s.c)))>eps&&vlen(xmult(subt(p,s.c),subt(p,s.a)))>eps;

}

int dot\_inplane\_ex(point3 p,point3 s1,point3 s2,point3 s3){

return dot\_inplane\_in(p,s1,s2,s3)&&vlen(xmult(subt(p,s1),subt(p,s2)))>eps&&

vlen(xmult(subt(p,s2),subt(p,s3)))>eps&&vlen(xmult(subt(p,s3),subt(p,s1)))>eps;

}

判两点在线段同侧

//点在线段上返回0,不共面无意义

int same\_side(point3 p1,point3 p2,line3 l){

return dmult(xmult(subt(l.a,l.b),subt(p1,l.b)),xmult(subt(l.a,l.b),subt(p2,l.b)))>eps;

}

int same\_side(point3 p1,point3 p2,point3 l1,point3 l2){

return dmult(xmult(subt(l1,l2),subt(p1,l2)),xmult(subt(l1,l2),subt(p2,l2)))>eps;

}

//判两点在平面同侧,点在平面上返回0

int same\_side(point3 p1,point3 p2,plane3 s){

return dmult(pvec(s),subt(p1,s.a))\*dmult(pvec(s),subt(p2,s.a))>eps;

}

int same\_side(point3 p1,point3 p2,point3 s1,point3 s2,point3 s3){

return dmult(pvec(s1,s2,s3),subt(p1,s1))\*dmult(pvec(s1,s2,s3),subt(p2,s1))>eps;

}

判两点在线段异侧

//点在线段上返回0,不共面无意义

int opposite\_side(point3 p1,point3 p2,line3 l){

return dmult(xmult(subt(l.a,l.b),subt(p1,l.b)),xmult(subt(l.a,l.b),subt(p2,l.b)))<-eps;

}

int opposite\_side(point3 p1,point3 p2,point3 l1,point3 l2){

return dmult(xmult(subt(l1,l2),subt(p1,l2)),xmult(subt(l1,l2),subt(p2,l2)))<-eps;

}

//点在平面上返回0

int opposite\_side(point3 p1,point3 p2,plane3 s){

return dmult(pvec(s),subt(p1,s.a))\*dmult(pvec(s),subt(p2,s.a))<-eps;

}

int opposite\_side(point3 p1,point3 p2,point3 s1,point3 s2,point3 s3){

return dmult(pvec(s1,s2,s3),subt(p1,s1))\*dmult(pvec(s1,s2,s3),subt(p2,s1))<-eps;

}

判两直线平行

int parallel(line3 u,line3 v){

return vlen(xmult(subt(u.a,u.b),subt(v.a,v.b)))<eps;

}

int parallel(point3 u1,point3 u2,point3 v1,point3 v2){

return vlen(xmult(subt(u1,u2),subt(v1,v2)))<eps;

}

判两平面平行

int parallel(plane3 u,plane3 v){

return vlen(xmult(pvec(u),pvec(v)))<eps;

}

int parallel(point3 u1,point3 u2,point3 u3,point3 v1,point3 v2,point3 v3){

return vlen(xmult(pvec(u1,u2,u3),pvec(v1,v2,v3)))<eps;

}

判直线与平面平行

int parallel(line3 l,plane3 s){

return zero(dmult(subt(l.a,l.b),pvec(s)));

}

int parallel(point3 l1,point3 l2,point3 s1,point3 s2,point3 s3){

return zero(dmult(subt(l1,l2),pvec(s1,s2,s3)));

}

判两直线垂直

int perpendicular(line3 u,line3 v){

return zero(dmult(subt(u.a,u.b),subt(v.a,v.b)));

}

int perpendicular(point3 u1,point3 u2,point3 v1,point3 v2){

return zero(dmult(subt(u1,u2),subt(v1,v2)));

}

判两平面垂直

int perpendicular(plane3 u,plane3 v){

return zero(dmult(pvec(u),pvec(v)));

}

int perpendicular(point3 u1,point3 u2,point3 u3,point3 v1,point3 v2,point3 v3){

return zero(dmult(pvec(u1,u2,u3),pvec(v1,v2,v3)));

}

判直线与平面平行

int perpendicular(line3 l,plane3 s){

return vlen(xmult(subt(l.a,l.b),pvec(s)))<eps;

}

int perpendicular(point3 l1,point3 l2,point3 s1,point3 s2,point3 s3){

return vlen(xmult(subt(l1,l2),pvec(s1,s2,s3)))<eps;

}

判两线段相交

//包括端点和部分重合

int intersect\_in(line3 u,line3 v){

if (!dots\_onplane(u.a,u.b,v.a,v.b))

return 0;

if (!dots\_inline(u.a,u.b,v.a)||!dots\_inline(u.a,u.b,v.b))

return !same\_side(u.a,u.b,v)&&!same\_side(v.a,v.b,u);

return dot\_online\_in(u.a,v)||dot\_online\_in(u.b,v)||dot\_online\_in(v.a,u)||dot\_online\_in(v.b,u);

}

int intersect\_in(point3 u1,point3 u2,point3 v1,point3 v2){

if (!dots\_onplane(u1,u2,v1,v2))

return 0;

if (!dots\_inline(u1,u2,v1)||!dots\_inline(u1,u2,v2))

return !same\_side(u1,u2,v1,v2)&&!same\_side(v1,v2,u1,u2);

return dot\_online\_in(u1,v1,v2)||dot\_online\_in(u2,v1,v2)||dot\_online\_in(v1,u1,u2)||dot\_online\_in(v2,u1,u2);

}

//不包括端点和部分重合

int intersect\_ex(line3 u,line3 v){

return dots\_onplane(u.a,u.b,v.a,v.b)&&opposite\_side(u.a,u.b,v)&&opposite\_side(v.a,v.b,u);

}

int intersect\_ex(point3 u1,point3 u2,point3 v1,point3 v2){

return dots\_onplane(u1,u2,v1,v2)&&opposite\_side(u1,u2,v1,v2)&&opposite\_side(v1,v2,u1,u2);

}

判线段与空间三角形相交

//包括交于边界和(部分)包含

int intersect\_in(line3 l,plane3 s){

return !same\_side(l.a,l.b,s)&&!same\_side(s.a,s.b,l.a,l.b,s.c)&&

!same\_side(s.b,s.c,l.a,l.b,s.a)&&!same\_side(s.c,s.a,l.a,l.b,s.b);

}

int intersect\_in(point3 l1,point3 l2,point3 s1,point3 s2,point3 s3){

return !same\_side(l1,l2,s1,s2,s3)&&!same\_side(s1,s2,l1,l2,s3)&&

!same\_side(s2,s3,l1,l2,s1)&&!same\_side(s3,s1,l1,l2,s2);

}

//不包括交于边界和(部分)包含

int intersect\_ex(line3 l,plane3 s){

return opposite\_side(l.a,l.b,s)&&opposite\_side(s.a,s.b,l.a,l.b,s.c)&&

opposite\_side(s.b,s.c,l.a,l.b,s.a)&&opposite\_side(s.c,s.a,l.a,l.b,s.b);

}

int intersect\_ex(point3 l1,point3 l2,point3 s1,point3 s2,point3 s3){

return opposite\_side(l1,l2,s1,s2,s3)&&opposite\_side(s1,s2,l1,l2,s3)&&

opposite\_side(s2,s3,l1,l2,s1)&&opposite\_side(s3,s1,l1,l2,s2);

}

计算两直线交点

//注意事先判断直线是否共面和平行!

//线段交点请另外判线段相交(同时还是要判断是否平行!)

point3 intersection(line3 u,line3 v){

point3 ret=u.a;

double t=((u.a.x-v.a.x)\*(v.a.y-v.b.y)-(u.a.y-v.a.y)\*(v.a.x-v.b.x))

/((u.a.x-u.b.x)\*(v.a.y-v.b.y)-(u.a.y-u.b.y)\*(v.a.x-v.b.x));

ret.x+=(u.b.x-u.a.x)\*t;

ret.y+=(u.b.y-u.a.y)\*t;

ret.z+=(u.b.z-u.a.z)\*t;

return ret;

}

point3 intersection(point3 u1,point3 u2,point3 v1,point3 v2){

point3 ret=u1;

double t=((u1.x-v1.x)\*(v1.y-v2.y)-(u1.y-v1.y)\*(v1.x-v2.x))

/((u1.x-u2.x)\*(v1.y-v2.y)-(u1.y-u2.y)\*(v1.x-v2.x));

ret.x+=(u2.x-u1.x)\*t;

ret.y+=(u2.y-u1.y)\*t;

ret.z+=(u2.z-u1.z)\*t;

return ret;

}

计算直线与平面交点

//注意事先判断是否平行,并保证三点不共线!

//线段和空间三角形交点请另外判断

point3 intersection(line3 l,plane3 s){

point3 ret=pvec(s);

double t=(ret.x\*(s.a.x-l.a.x)+ret.y\*(s.a.y-l.a.y)+ret.z\*(s.a.z-l.a.z))/

(ret.x\*(l.b.x-l.a.x)+ret.y\*(l.b.y-l.a.y)+ret.z\*(l.b.z-l.a.z));

ret.x=l.a.x+(l.b.x-l.a.x)\*t;

ret.y=l.a.y+(l.b.y-l.a.y)\*t;

ret.z=l.a.z+(l.b.z-l.a.z)\*t;

return ret;

}

point3 intersection(point3 l1,point3 l2,point3 s1,point3 s2,point3 s3){

point3 ret=pvec(s1,s2,s3);

double t=(ret.x\*(s1.x-l1.x)+ret.y\*(s1.y-l1.y)+ret.z\*(s1.z-l1.z))/

(ret.x\*(l2.x-l1.x)+ret.y\*(l2.y-l1.y)+ret.z\*(l2.z-l1.z));

ret.x=l1.x+(l2.x-l1.x)\*t;

ret.y=l1.y+(l2.y-l1.y)\*t;

ret.z=l1.z+(l2.z-l1.z)\*t;

return ret;

}

计算两平面交线

//注意事先判断是否平行,并保证三点不共线!

line3 intersection(plane3 u,plane3 v){

line3 ret;

ret.a=parallel(v.a,v.b,u.a,u.b,u.c)?intersection(v.b,v.c,u.a,u.b,u.c):intersection(v.a,v.b,u.a,u.b,u.c);

ret.b=parallel(v.c,v.a,u.a,u.b,u.c)?intersection(v.b,v.c,u.a,u.b,u.c):intersection(v.c,v.a,u.a,u.b,u.c);

return ret;

}

line3 intersection(point3 u1,point3 u2,point3 u3,point3 v1,point3 v2,point3 v3){

line3 ret;

ret.a=parallel(v1,v2,u1,u2,u3)?intersection(v2,v3,u1,u2,u3):intersection(v1,v2,u1,u2,u3);

ret.b=parallel(v3,v1,u1,u2,u3)?intersection(v2,v3,u1,u2,u3):intersection(v3,v1,u1,u2,u3);

return ret;

}

点到直线距离

double ptoline(point3 p,line3 l){

return vlen(xmult(subt(p,l.a),subt(l.b,l.a)))/distance(l.a,l.b);

}

double ptoline(point3 p,point3 l1,point3 l2){

return vlen(xmult(subt(p,l1),subt(l2,l1)))/distance(l1,l2);

}

点到平面距离

double ptoplane(point3 p,plane3 s){

return fabs(dmult(pvec(s),subt(p,s.a)))/vlen(pvec(s));

}

double ptoplane(point3 p,point3 s1,point3 s2,point3 s3){

return fabs(dmult(pvec(s1,s2,s3),subt(p,s1)))/vlen(pvec(s1,s2,s3));

}

直线到直线距离

double linetoline(line3 u,line3 v){

point3 n=xmult(subt(u.a,u.b),subt(v.a,v.b));

return fabs(dmult(subt(u.a,v.a),n))/vlen(n);

}

double linetoline(point3 u1,point3 u2,point3 v1,point3 v2){

point3 n=xmult(subt(u1,u2),subt(v1,v2));

return fabs(dmult(subt(u1,v1),n))/vlen(n);

}

两直线夹角cos值

double angle\_cos(line3 u,line3 v){

return dmult(subt(u.a,u.b),subt(v.a,v.b))/vlen(subt(u.a,u.b))/vlen(subt(v.a,v.b));

}

double angle\_cos(point3 u1,point3 u2,point3 v1,point3 v2){

return dmult(subt(u1,u2),subt(v1,v2))/vlen(subt(u1,u2))/vlen(subt(v1,v2));

}

两平面夹角cos值

double angle\_cos(plane3 u,plane3 v){

return dmult(pvec(u),pvec(v))/vlen(pvec(u))/vlen(pvec(v));

}

double angle\_cos(point3 u1,point3 u2,point3 u3,point3 v1,point3 v2,point3 v3){

return dmult(pvec(u1,u2,u3),pvec(v1,v2,v3))/vlen(pvec(u1,u2,u3))/vlen(pvec(v1,v2,v3));

}

直线平面夹角sin值

double angle\_sin(line3 l,plane3 s){

return dmult(subt(l.a,l.b),pvec(s))/vlen(subt(l.a,l.b))/vlen(pvec(s));

}

double angle\_sin(point3 l1,point3 l2,point3 s1,point3 s2,point3 s3){

return dmult(subt(l1,l2),pvec(s1,s2,s3))/vlen(subt(l1,l2))/vlen(pvec(s1,s2,s3));

}

构造凸包

//graham算法O(nlogn)

MyPoint p1, p2;

inline bool graham\_less(const MyPoint& a, const MyPoint& b) {

double ret = xmult(a, b, p1);

if (zero(ret)) {

return xmult(a, b, p2) < 0;

} else {

return ret < 0;

}

}

void \_graham(int n, MyPoint\* p, int& s, MyPoint\* ch) {

int i, k = 0;

for (p1 = p2 = p[0], i = 1; i < n; p2.x += p[i].x, p2.y += p[i].y, i++) {

if (p1.y - p[i].y > eps || (zero(p1.y - p[i].y) && p1.x > p[i].x)) {

p1 = p[k = i];

}

}

p2.x /= n, p2.y /= n;

p[k] = p[0], p[0] = p1;

sort(p + 1, p + n, graham\_less);

for (ch[0] = p[0], ch[1] = p[1], ch[2] = p[2], s = i = 3; i < n; ch[s++] =

p[i++]) {

for (; s > 2 && xmult(ch[s - 2], p[i], ch[s - 1]) < -eps; s--) {

;

}

}

}

//构造凸包接口函数,传入原始点集大小n,点集p(p原有顺序会被打乱!)

//返回凸包大小,凸包的点在convex中

//参数maxsize为1包含共线点,为0不包含共线点，参数clockwise为1顺时针构造,为0逆时针构造

//在输入仅有若干共线点时算法不稳定,可能有此类情况请另行处理!

//不能去掉点集中重合的点

int graham(int n, MyPoint\* p, MyPoint\* convex, int maxsize = 1, int dir = 1) {

MyPoint\* temp = new MyPoint[n];

int s, i;

\_graham(n, p, s, temp);

convex[0] = temp[0], n = 1, i = (dir ? 1 : (s - 1));

for (; dir ? (i < s) : i; i += (dir ? 1 : -1) ) {

if (maxsize || !zero(xmult(temp[i-1],temp[i],temp[(i+1)%s]))) {

convex[n++]=temp[i];

}

}

delete[] temp;

return n;

}

网格

//公共头

多边形上的网格点个数

int grid\_onedge(int n,point\* p){

int i,ret=0;

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

ret+=gcd(fabs(p[i].x-p[(i+1)%n].x),fabs(p[i].y-p[(i+1)%n].y));

return ret;

}

多边形内的网格点个数

int grid\_inside(int n,point\* p){

int i,ret=0;

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

ret+=p[(i+1)%n].y\*(p[i].x-p[(i+2)%n].x);

return (abs(ret)-grid\_onedge(n,p))/2+1;

}

圆

double disptoline(point p,point l1,point l2){

return fabs(xmult(p,l1,l2))/distance(l1,l2);

}

point intersection(point u1,point u2,point v1,point v2){

point ret=u1;

double t=((u1.x-v1.x)\*(v1.y-v2.y)-(u1.y-v1.y)\*(v1.x-v2.x))

/((u1.x-u2.x)\*(v1.y-v2.y)-(u1.y-u2.y)\*(v1.x-v2.x));

ret.x+=(u2.x-u1.x)\*t;

ret.y+=(u2.y-u1.y)\*t;

return ret;

}

判直线和圆相交

//包括相切

int intersect\_line\_circle(point c,double r,point l1,point l2){

return disptoline(c,l1,l2)<r+eps;

}

判线段和圆相交

//包括端点和相切

int intersect\_seg\_circle(point c,double r,point l1,point l2){

double t1=distance(c,l1)-r,t2=distance(c,l2)-r;

point t=c;

if (t1<eps||t2<eps)

return t1>-eps||t2>-eps;

t.x+=l1.y-l2.y;

t.y+=l2.x-l1.x;

return xmult(l1,c,t)\*xmult(l2,c,t)<eps&&disptoline(c,l1,l2)-r<eps;

}

判圆和圆相交,包括相切

int intersect\_circle\_circle(point c1,double r1,point c2,double r2){

return distance(c1,c2)<r1+r2+eps&&distance(c1,c2)>fabs(r1-r2)-eps;

}

计算圆上到点p最近点

//如p与圆心重合,返回p本身

point dot\_to\_circle(point c,double r,point p){

point u,v;

if (distance(p,c)<eps)

return p;

u.x=c.x+r\*fabs(c.x-p.x)/distance(c,p);

u.y=c.y+r\*fabs(c.y-p.y)/distance(c,p)\*((c.x-p.x)\*(c.y-p.y)<0?-1:1);

v.x=c.x-r\*fabs(c.x-p.x)/distance(c,p);

v.y=c.y-r\*fabs(c.y-p.y)/distance(c,p)\*((c.x-p.x)\*(c.y-p.y)<0?-1:1);

return distance(u,p)<distance(v,p)?u:v;

}

计算直线与圆的交点

//保证直线与圆有交点

//计算线段与圆的交点可用这个函数后判点是否在线段上

void intersection\_line\_circle(point c,double r,point l1,point l2,point& p1,point& p2){

point p=c;

double t;

p.x+=l1.y-l2.y;

p.y+=l2.x-l1.x;

p=intersection(p,c,l1,l2);

t=sqrt(r\*r-distance(p,c)\*distance(p,c))/distance(l1,l2);

p1.x=p.x+(l2.x-l1.x)\*t;

p1.y=p.y+(l2.y-l1.y)\*t;

p2.x=p.x-(l2.x-l1.x)\*t;

p2.y=p.y-(l2.y-l1.y)\*t;

}

计算圆与圆的交点

//保证圆与圆有交点,圆心不重合

void intersection\_circle\_circle(point c1,double r1,point c2,double r2,point& p1,point& p2){

point u,v;

double t;

t=(1+(r1\*r1-r2\*r2)/distance(c1,c2)/distance(c1,c2))/2;

u.x=c1.x+(c2.x-c1.x)\*t;

u.y=c1.y+(c2.y-c1.y)\*t;

v.x=u.x+c1.y-c2.y;

v.y=u.y-c1.x+c2.x;

intersection\_line\_circle(c1,r1,u,v,p1,p2);

}

整数函数

公共头

//注意某些情况下整数运算会出界!

#define sign(a) ((a)>0?1:(((a)<0?-1:0)))

struct point{int x,y;};

struct line{point a,b;};

判三点共线

int dots\_inline(point p1,point p2,point p3){

return !xmult(p1,p2,p3);

}

int dots\_inline(int x1,int y1,int x2,int y2,int x3,int y3){

return !xmult(x1,y1,x2,y2,x3,y3);

}

判点是否在线段上

//包括端点和部分重合

int dot\_online\_in(point p,line l){

return !xmult(p,l.a,l.b)&&(l.a.x-p.x)\*(l.b.x-p.x)<=0&&(l.a.y-p.y)\*(l.b.y-p.y)<=0;

}

int dot\_online\_in(point p,point l1,point l2){

return !xmult(p,l1,l2)&&(l1.x-p.x)\*(l2.x-p.x)<=0&&(l1.y-p.y)\*(l2.y-p.y)<=0;

}

int dot\_online\_in(int x,int y,int x1,int y1,int x2,int y2){

return !xmult(x,y,x1,y1,x2,y2)&&(x1-x)\*(x2-x)<=0&&(y1-y)\*(y2-y)<=0;

}

//不包括端点

int dot\_online\_ex(point p,line l){

return dot\_online\_in(p,l)&&(p.x!=l.a.x||p.y!=l.a.y)&&(p.x!=l.b.x||p.y!=l.b.y);

}

int dot\_online\_ex(point p,point l1,point l2){

return dot\_online\_in(p,l1,l2)&&(p.x!=l1.x||p.y!=l1.y)&&(p.x!=l2.x||p.y!=l2.y);

}

int dot\_online\_ex(int x,int y,int x1,int y1,int x2,int y2){

return dot\_online\_in(x,y,x1,y1,x2,y2)&&(x!=x1||y!=y1)&&(x!=x2||y!=y2);

}

判两点在直线同侧

//点在直线上返回0

int same\_side(point p1,point p2,line l){

return sign(xmult(l.a,p1,l.b))\*xmult(l.a,p2,l.b)>0;

}

int same\_side(point p1,point p2,point l1,point l2){

return sign(xmult(l1,p1,l2))\*xmult(l1,p2,l2)>0;

}

判两点在直线异侧

//点在直线上返回0

int opposite\_side(point p1,point p2,line l){

return sign(xmult(l.a,p1,l.b))\*xmult(l.a,p2,l.b)<0;

}

int opposite\_side(point p1,point p2,point l1,point l2){

return sign(xmult(l1,p1,l2))\*xmult(l1,p2,l2)<0;

}

判两直线平行

int parallel(line u,line v){

return (u.a.x-u.b.x)\*(v.a.y-v.b.y)==(v.a.x-v.b.x)\*(u.a.y-u.b.y);

}

int parallel(point u1,point u2,point v1,point v2){

return (u1.x-u2.x)\*(v1.y-v2.y)==(v1.x-v2.x)\*(u1.y-u2.y);

}

判两直线垂直

int perpendicular(line u,line v){

return (u.a.x-u.b.x)\*(v.a.x-v.b.x)==-(u.a.y-u.b.y)\*(v.a.y-v.b.y);

}

int perpendicular(point u1,point u2,point v1,point v2){

return (u1.x-u2.x)\*(v1.x-v2.x)==-(u1.y-u2.y)\*(v1.y-v2.y);

}

判两线段相交

//包括端点和部分重合

int intersect\_in(line u,line v){

if (!dots\_inline(u.a,u.b,v.a)||!dots\_inline(u.a,u.b,v.b))

return !same\_side(u.a,u.b,v)&&!same\_side(v.a,v.b,u);

return dot\_online\_in(u.a,v)||dot\_online\_in(u.b,v)||dot\_online\_in(v.a,u)||dot\_online\_in(v.b,u);

}

int intersect\_in(point u1,point u2,point v1,point v2){

if (!dots\_inline(u1,u2,v1)||!dots\_inline(u1,u2,v2))

return !same\_side(u1,u2,v1,v2)&&!same\_side(v1,v2,u1,u2);

return dot\_online\_in(u1,v1,v2)||dot\_online\_in(u2,v1,v2)||dot\_online\_in(v1,u1,u2)||dot\_online\_in(v2,u1,u2);

}

//不包括端点和部分重合

int intersect\_ex(line u,line v){

return opposite\_side(u.a,u.b,v)&&opposite\_side(v.a,v.b,u);

}

int intersect\_ex(point u1,point u2,point v1,point v2){

return opposite\_side(u1,u2,v1,v2)&&opposite\_side(v1,v2,u1,u2);

}

旋转卡壳法求凸包直径

int rotating\_calipers(Xpoint \*ch, int n)

{

int q = 1, ans = 0;

ch[n] = ch[0];

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

{

while (cross(ch[p + 1], ch[q + 1], ch[p]) > cross(ch[p + 1], ch[q], ch[p]))

q = (q + 1) % n;

ans = max(ans, max(dist2(ch[p], ch[q]), dist2(ch[p + 1], ch[q + 1])));

}

return ans;

}

是否是凸包（要求逆时针）

bool convex()

{

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

if (dblcmp(xmult(point[(i + 1) % n] - point[i], point[(i + 2) % n] - point[(i + 1) % n])) < 0)

return false;

return true;

}

点是否在凸包内（要求逆时针）

bool inconvex(Point &peg)

{

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

if (dblcmp(xmult(point[(i + 1) % n] - point[i], peg - point[(i + 1) % n])) <= 0)

return false;

return true;

}

欧拉回路（无向图）

#define nmax 1001

int n, m, mark[nmax], num[nmax], map[nmax][nmax];

void dfs(int s) {

mark[s] = 0;

int i;

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

if (map[s][i] && mark[i]) {

dfs(i);

}

}

}

void solve() {

int i;

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

if (num[i] & 1) {

puts("no");

return;

}

}

dfs(1);

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

if (mark[i]) {

puts("no");

return;

}

}

puts("yes");

}

int main() {

int t, i, j, a, b;

while (scanf("%d", &t) != EOF) {

while (t--) {

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

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

for (j = 0; j <= n; j++) {

map[i][j] = 0;

}

mark[i] = 1;

num[i] = 0;

}

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

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

map[a][b] = 1, map[b][a] = 1;

num[a]++, num[b]++;

}

solve();

}

}

return 0;

}

点在多边形内（转角法）

#define nmax 110

#define pi acos(-1.0)

#define eps 1e-8

typedef struct point {

double x, y;

} point;

point Point[nmax], p, a, b;

double cross\_product(point p0, point p1, point p2) {

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

}

double dot\_product(point p0, point p1, point p2) {

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

}

double pDistans(point n, point m) {

return ((n.x - m.x) \* (n.x - m.x) + (n.y - m.y) \* (n.y - m.y));

}

void solve(int n) {

if (n < 3) {

puts("no");

return;

}

int i;

double temp, res;

for (i = 0, res = 0.0; i < n; i++) {

a = Point[i], b = Point[(i + 1) % n];

temp = cross\_product(p, a, b);

if (temp > 0) {

res += acos(

dot\_product(p, a, b)

/ sqrt(pDistans(p, a) \* pDistans(p, b)));

} else {

res -= acos(

dot\_product(p, a, b)

/ sqrt(pDistans(p, a) \* pDistans(p, b)));

}

}

res = fabs(res / pi) - 2.0;

if (fabs(res) < eps) {

puts("yes");

} else {

puts("no");

}

}

int main() {

int t, i, n;

while (scanf("%d", &t) != EOF) {

while (t--) {

scanf("%d", &n);

scanf("%lf %lf", &p.x, &p.y);

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

scanf("%lf %lf", &Point[i].x, &Point[i].y);

}

solve(n);

}

}

return 0;

}

Pick公式

//整点多边形的面积=内部整点个数+边上的整点个数/2 - 1

struct Point {

int x; int y;

};

//叉积

int VectorProduct(Point a, Point b, Point c) {

return (a.x - b.x) \* (a.y - c.y) - (a.x - c.x) \* (a.y - b.y);

}

int Gcd(int x, int y) {

if (x < 0)

x = -x;

if (y < 0)

y = -y;

int r = 1;

if (y == 0)

return x;

while (r) {

r = x % y;

if (r) {

x = y; y = r;

}

}

return y;

}

int main() {

Point point[3];

int area, numpoint, res;

while (scanf("%d %d %d %d %d %d", &point[0].x, &point[0].y, &point[1].x,

&point[1].y, &point[2].x, &point[2].y) != EOF) {

if (point[0].x == 0 && point[0].y == 0 && point[1].x == 0 && point[1].y

== 0 && point[2].x == 0 && point[2].y == 0)

break;

area = abs(VectorProduct(point[0], point[1], point[2])) / 2;

numpoint = Gcd(point[0].x - point[1].x, point[0].y - point[1].y) + Gcd(

point[0].x - point[2].x, point[0].y - point[2].y) + Gcd(

point[1].x - point[2].x, point[1].y - point[2].y);

res = area - numpoint / 2 + 1;

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

}

return 0;

}

极坐标排序与二分查找

//天津2010网络赛题

const int MAXN = 705;

const double PI = acos(-1);

typedef struct {

int x; int y;

} MyPoint;

int N;

MyPoint points[MAXN];

double angs[MAXN][MAXN];

int findindex(double \*angarray, double ang) {

int low, mid, high;

low = 0;

high = N - 2;

while (low <= high) {

mid = (low + high) / 2;

if (angarray[mid] > ang) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return low;

}

void work() {

int T, i, j, tempindex, center;

long long tu, ao, num, tempnum;

double oppang;

scanf("%d", &T);

while (T--) {

scanf("%d", &N);

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

scanf("%d%d", &points[i].x, &points[i].y);

}

tu = ((long long) N) \* (N - 1) \* (N - 2) \* (N - 3) / 24;

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

tempindex = 0;

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

if (j != i) {

angs[i][tempindex++] = atan2(points[j].y - points[i].y,

points[j].x - points[i].x);

}

}

sort(angs[i], angs[i] + N - 1);

}

ao = ((long long) (N - 1)) \* (N - 2) \* (N - 3) / 6;

for (center = 0; center < N; center++) {

tempnum = 0;

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

if (angs[center][i] > 0) {

oppang = angs[center][i] - PI;

} else {

oppang = angs[center][i] + PI;

}

j = findindex(angs[center], oppang);

num = (j + N - i - 2) % (N - 1);

tempnum += num \* (num - 1) / 2;

}

tu -= ao - tempnum;

}

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

}

}

int main() {

work();

return 0;

}

1. 其它

高精度样例-大数阶乘函数

void dashujiecheng(int N) {

int a[10000]={1,0},temp,i,j,k;

int X=1;

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

for(j=0;j<X;j++) {

a[j]\*=i;

}

for(k=0;k<X;k++) {

if(a[k]>=10000) {

if(X==k+1)

X++;

temp=a[k]%10000;

a[k+1]+=(a[k]-temp)/10000;

a[k]=temp;

}

}

}

cout<<a[X-1];

for(k=X-2;k>=0;k--) {

if(a[k]<1000)

cout<<0;

if(a[k]<100)

cout<<0;

if(a[k]<10)

cout<<0;

cout<<a[k];

}

cout<<endl;

}

递归下降法样例-表达式求值

//杭电4140题几乎完整代码，

typedef long long LL;

int x, cef,idx; bool sg; LL res;

LL pw(int x, int a){

LL d = 1, b = x;

while (a) {

if (a & 1) d \*= b;

b \*= b; a >>= 1;

}

return d;

}

char peek(){

char c = getchar();

ungetc(c, stdin);

return c;

}

void skip(){

int c = getchar();

while(c <= ' ') { c = getchar();}

ungetc(c, stdin);

}

bool expect(int ch){

if (peek() == ch) {

getchar(); return true;

}

return false;

}

void sign\_(){

if (expect('-')) {

sg = false;

} else {

expect('+'); sg = true;

}

}

void coef\_(){

if (peek() >= '0' && peek() <= '9')

scanf("%d", &cef);

else

cef = 1;

}

void index\_(){

if (expect('X')) {

if (expect('^')) {

scanf("%d", &idx);

} else {

idx = 1;

}

} else {

idx = 0;

}

}

void term(){

sign\_();

coef\_();

index\_();

res += (sg ? cef : -cef) \* pw(x, idx);

}

void run(){

char ch;

res = 0;

skip();

do {

term();

ch = peek();

} while (ch == '+' || ch == '-');

}

int main() {

int T;

scanf("%d", &T);

for(int t = 1; t <= T; t++) {

scanf("%d", &x);

run();

printf("Case #%d: %I64d\n", t, res);

}

return 0;

}

若干STL数据结构用法

优先队列先弹出小整数的定义方法priority\_queue<int,vector<int>,greater<int> >

其中greater<int>为一个结构体模板，形式如下

struct greater {

bool

operator()(const int &i1, const int &i2) const

{ return i1 < i2; }

};

可按此形式写出自定义的比较方法

Copy函数的用法举例

int myarray[20] = {6, 1, 3, 5, 2};

set<int> myset;

copy(myarray, myarray + 5, inserter(myset, myset.begin()));

vector<int> myvector;

copy(myset.begin(), myset.end(), inserter(myvector, myvector.begin()));

copy(myvector.begin(), myvector.end(), ostream\_iterator<int>(cout, "\t"));

如上代码可以输出1 2 3 5 6

Java正则表达式用法样例

public static void main(String[] args) {

String str;

Pattern p = Pattern.compile("long long", Pattern.CASE\_INSENSITIVE);

Matcher m;

while (cin.hasNextLine()) {

str = cin.nextLine();

while (true) {

m = p.matcher(str);

if(!m.find()){

break;

}

str = m.replaceFirst("LL");

}

System.out.println(str);

}

}

1. 国内外著名OJ简介

北大OJ

杭电OJ

浙大OJ

1. GNU C与ANSI C的区别

有些编译器不支持strrev函数