****

**fudq**

**Beijing Forestry University**

**bjfudq@gmail.com**

**2015/10/16**

**ACM/ICPC常用算法模板**

目 录

[一、小助手 1](#_Toc432769545)

[头文件及宏定义 1](#_Toc432769546)

[输入外挂 1](#_Toc432769547)

[Java模板 2](#_Toc432769548)

[二、STL 3](#_Toc432769549)

[vector & iterate 3](#_Toc432769550)

[map 3](#_Toc432769551)

[queue 4](#_Toc432769552)

[stack 4](#_Toc432769553)

[priority\_queue 5](#_Toc432769554)

[set 5](#_Toc432769555)

[list 7](#_Toc432769556)

[string 8](#_Toc432769557)

[next\_permutation 9](#_Toc432769558)

[Others 9](#_Toc432769559)

[三、组合数学 10](#_Toc432769560)

[Lucas定理 10](#_Toc432769561)

[互质数最大不能组合数 10](#_Toc432769562)

[母函数 11](#_Toc432769563)

[容斥原理 11](#_Toc432769564)

[整数划分 12](#_Toc432769565)

[组合数打表 13](#_Toc432769566)

[组合数普通求法 13](#_Toc432769567)

[四、数论 14](#_Toc432769568)

[Baby Step Giant Step(n不为素数) 14](#_Toc432769569)

[Baby Step Giant Step(n为素数) 15](#_Toc432769570)

[lattice-count栅格计数问题 15](#_Toc432769571)

[Stirling数应用 15](#_Toc432769572)

[乘法逆元 16](#_Toc432769573)

[大数模板 16](#_Toc432769574)

[大素数判断(miller-rabin算法)和质因数分解(Pollard\_rho算法) 20](#_Toc432769575)

[多维曼哈顿距离 22](#_Toc432769576)

[费马小定理与素数测试 23](#_Toc432769577)

[傅里叶快速变换求两数乘积 24](#_Toc432769578)

[高次幂取模 25](#_Toc432769579)

[高斯消元（浮点数） 25](#_Toc432769580)

[矩阵乘法 25](#_Toc432769581)

[矩阵快速幂 26](#_Toc432769582)

[卡特兰数 26](#_Toc432769583)

[快速乘取余a\*b%c 27](#_Toc432769584)

[快速幂取余a^b%c(蒙哥马利算法) 27](#_Toc432769585)

[扩展欧几里德 27](#_Toc432769586)

[拉宾-米勒测试 27](#_Toc432769587)

[牛顿迭代法求开方 28](#_Toc432769588)

[欧拉函数 28](#_Toc432769589)

[判断n(最大10^18)是否能被一个数的平方整除 29](#_Toc432769590)

[求1到n的n个数的最小公倍数 29](#_Toc432769591)

[求解x^2=a mod (n) 30](#_Toc432769592)

[求佩尔方程的第k解 30](#_Toc432769593)

[日期相关函数 31](#_Toc432769594)

[数学公式 32](#_Toc432769595)

[素数判定 32](#_Toc432769596)

[素数筛选法 32](#_Toc432769597)

[(Eratosthenes埃拉托斯特尼筛选法) 32](#_Toc432769598)

[线性同余式方程 32](#_Toc432769599)

[质因数分解 33](#_Toc432769600)

[质因数分解求n个数的最小公倍数 33](#_Toc432769601)

[质因数分解求大数阶乘对p取余 34](#_Toc432769602)

[中国剩余定理 35](#_Toc432769603)

[最大公约数和最小公倍数 35](#_Toc432769604)

[五、博弈 36](#_Toc432769605)

[Crazy Nim 36](#_Toc432769606)

[Every-SG游戏 36](#_Toc432769607)

[SG函数 36](#_Toc432769608)

[博弈搜索树 37](#_Toc432769609)

[取火柴游戏 38](#_Toc432769610)

[三类经典博弈 39](#_Toc432769611)

[六、字符串 41](#_Toc432769612)

[KMP 41](#_Toc432769613)

[AC自动机 41](#_Toc432769614)

[Manacher算法求字符串 43](#_Toc432769615)

[最长回文子串 43](#_Toc432769616)

[Trie树求字符串所有不相同子串 43](#_Toc432769617)

[后缀数组 44](#_Toc432769618)

[后缀自动机 45](#_Toc432769619)

[扩展KMP 46](#_Toc432769620)

[七、图论 49](#_Toc432769621)

[tarjan求带重边无向图的割边 49](#_Toc432769622)

[并查集 49](#_Toc432769623)

[最大团 50](#_Toc432769624)

[最小生成树 51](#_Toc432769625)

[最短路-Dijkstra 52](#_Toc432769626)

[最短路-Bellman-Ford 52](#_Toc432769627)

[最短路-Floyed 53](#_Toc432769628)

[最短路-SPFA 53](#_Toc432769629)

[欧拉回路 54](#_Toc432769630)

[拓扑排序 55](#_Toc432769631)

[八、网络流 56](#_Toc432769632)

[二分图最大匹配-匈牙利算法 56](#_Toc432769633)

[完备匹配下最大权匹配-KM算法 56](#_Toc432769634)

[稳定婚姻问题 57](#_Toc432769635)

[九、树论 59](#_Toc432769636)

[LCA离线Tarjan算法 59](#_Toc432769637)

[LCA离线Tarjan算法-hdu2586 59](#_Toc432769638)

[LCA离线Tarjan算法-poj1470 60](#_Toc432769639)

[LCA在线算法 61](#_Toc432769640)

[求哈夫曼树的WPL 62](#_Toc432769641)

[求树的直径 63](#_Toc432769642)

[求树中每个结点的子树结点个数 63](#_Toc432769643)

[已知二叉树先序遍历和中序遍历 63](#_Toc432769644)

[求后序遍历 63](#_Toc432769645)

[十、数据结构 65](#_Toc432769646)

[RMQ 65](#_Toc432769647)

[树状数组 65](#_Toc432769648)

[线段树 66](#_Toc432769649)

[十一、动态规划 70](#_Toc432769650)

[背包九讲 70](#_Toc432769651)

[P01: 01背包问题 70](#_Toc432769652)

[P02: 完全背包问题 70](#_Toc432769653)

[P03: 多重背包问题 71](#_Toc432769654)

[P04: 混合三种背包问题 71](#_Toc432769655)

[P05: 二维费用的背包问题 72](#_Toc432769656)

[P06: 分组的背包问题 72](#_Toc432769657)

[P07: 有依赖的背包问题 72](#_Toc432769658)

[P08: 泛化物品 73](#_Toc432769659)

[P09: 背包问题问法的变化 73](#_Toc432769660)

[最长公共子序列 74](#_Toc432769661)

[最长递增子序列 75](#_Toc432769662)

[最长公共递增子序列 76](#_Toc432769663)

[最大子段和 76](#_Toc432769664)

[最大子块和 77](#_Toc432769665)

[记忆化搜索 77](#_Toc432769666)

[期望类dp 78](#_Toc432769667)

[状态压缩dp 79](#_Toc432769668)

[数位dp 82](#_Toc432769669)

[十二、搜索&查找 84](#_Toc432769670)

[bfs 84](#_Toc432769671)

[dfs 84](#_Toc432769672)

[hash 85](#_Toc432769673)

[常见字符串hash算法 86](#_Toc432769674)

[二分查找 87](#_Toc432769675)

[三分查找 88](#_Toc432769676)

[十三、排序 89](#_Toc432769677)

[快速排序 89](#_Toc432769678)

[归并排序 89](#_Toc432769679)

[希尔排序 89](#_Toc432769680)

[冒泡排序 89](#_Toc432769681)

[堆排序 89](#_Toc432769682)

[选择排序 90](#_Toc432769683)

[十四、计算几何 91](#_Toc432769684)

[计算几何完全模板 91](#_Toc432769685)

[计算几何完全模板（补充） 93](#_Toc432769686)

[判断两条线段是否相交 96](#_Toc432769687)

[求点到矩形的最短距离和最长距离 96](#_Toc432769688)

[求覆盖三点的最小圆 97](#_Toc432769689)

[求空间直线间距离 98](#_Toc432769690)

[求两个圆相交面积 98](#_Toc432769691)

[求两空间异面直线公垂线及交点坐标 98](#_Toc432769692)

[求三角形整数内点个数 99](#_Toc432769693)

[凸包 99](#_Toc432769694)

[已知球面两点经纬度求距离 100](#_Toc432769695)

[最近点对问题 100](#_Toc432769696)

### 一、小助手

头文件及宏定义

#include <functional>

#include <algorithm>

#include <iostream>

//#include <fstream>

#include <sstream>

#include <iomanip>

#include <numeric>

#include <cstring>

#include <cassert>

#include <cstdio>

#include <string>

#include <vector>

#include <bitset>

#include <queue>

#include <stack>

#include <cmath>

#include <ctime>

#include <list>

#include <set>

#include <map>

using namespace std;

//#pragma comment(linker,"/STACK:102400000,102400000")

#define MEM(a) (memset((a),0,sizeof(a)))

#define LEN(a) (int)strlen((a))

#define fr(a) for(int i=0;i<(a);i++)

#define sf(a) scanf("%d",&(a))

#define sf64(a) scanf("%I64d",&(a))

#define sff(a) scanf("%lf",&(a))

#define sfs(a) scanf("%s",(a))

#define sf2(a,b) scanf("%d%d",&(a),&(b))

#define sf3(a,b,c) scanf("%d%d%d",&(a),&(b),&(c))

#define sf2s(a,b) scanf("%s%s",(a),(b));

#define sf2f(a,b) scanf("%lf%lf",&(a),&(b))

#define sf3f(a,b,c) scanf("%lf%lf%lf",&(a),&(b),&(c))

#define sf264(a,b) scanf("%I64d%I64d",&(a),&(b))

#define sf364(a,b,c) scanf("%I64d%I64d%I64d",&(a),&(b),&(c))

#define pf(a) printf("%d\n",(a))

#define pfc(a) printf("%c",(a));

#define pf64(a) printf("%I64d\n",(a))

#define pff(a) printf("%f\n",(a))

#define pfs(a) printf("%s\n",(a))

#define pf2(a,b) printf("%d %d\n",(a),(b))

#define pf2s(a,b) printf("%s%s\n",(a),(b));

#define pf2f(a,b) printf("%f %f\n",(a),(b))

#define pf264(a,b) printf("%I64d %I64d\n",(a),(b))

#define pfn printf("\n")

#define pfk printf(" ")

#define LL long long

const int N=100010;

const int M=1000010;

const int MOD=1000000007;

const int INF=0x7fffffff;

const int dir4[4][2]={{-1,0},{1,0},{0,-1},{0,1}};

const int dir8[8][2]={{-1,0},{-1,1},{0,1},{1,1},{1,0},{1,-1},{0,-1},{-1,-1}};

const double eps=1e-8;

const double PI=acos(-1.0);

inline int sign(double x){return (x>eps)-(x<-eps);}

template<class T> T gcd(T a,T b){return b?gcd(b,a%b):a;}

template<class T> inline T Min(T a,T b){return a<b?a:b;}

template<class T> inline T Max(T a,T b){return a>b?a:b;}

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

int main()

{

#ifndef ONLINE\_JUDGE

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

// freopen("testout.txt", "w", stdout);

#endif

return 0;

}

输入外挂

//非负整数

int get\_int()

{

int ch,res=0;

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;

}

//整数

int get\_int2()

{

int res = 0, ch, flag = 0;

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

if (ch == '-')

flag = 1;

if (ch == EOF)

return -1;

}

res = ch - '0';

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

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

if (flag == 1)

res = -res;

return res;

}

Java模板

Biginteger的常见操作函数：

加减乘除函数：add,subtract,multiply,divide

取模函数：mod

比较数是否相等函数：equals,返回true或者false

比较数大小函数：compareTo,返回大于0，小于0，等于0

取较大值或者较小值函数：max,min

求幂乘函数：pow

求最大公约数函数：gcd

求绝对值函数：abs

/\*BigDecimal类的stripTrailingZeros()函数：去除结果的后缀0

BigDecimal类的toPlainString()函数与toString()函数区别:如果数很大很大，前者输出的是完整的结果，后者输出的是科学计数法

例：

BigDecimal b ;（b=（0.4321^20)

若String s = b.toPlainString() ;

System.out.println(s) ;

输出为：

0.00000005148554641076956121994511276767154838481760200726351203835429763013462401

若String s = b.toString() ;

输出为：

5.148554641076956121994511276767154838481760200726351203835429763013462401E-8

\*/

import java.io.\*;

import java.util.\*;

import java.math.\*;

public class Main

{

public static void main(String[] args)

{

Scanner cin=new Scanner(System.in);

String str;

BigInteger a,ans,t; //定义BigInteger类型和String类型

BigInteger base=BigInteger.valueOf(26); //base初始化为整数的26

BigInteger f[]=new BigInteger[110]; //定义大小为26的BigInteger数组

while(cin.hasNext()) //当有数据输入时

{

str=cin.next(); //读取一个String类型（BigInteger）的数据

t=cin.nextBigInteger();

int len=str.length(); //len代表str的长度

int i=5;

char temp=str.charAt(i); //取string类型的第i个字符

a=new BigInteger(str); //将String类型的str转成BigInteger类型的a

int p=Integer.parseInt(str,10); //将str转成十进制的整数p

char k=(char)(p); //将整数p转成char类型的k

str=ans.toString(); //将BigInteger类型的ans转成String类型的str

System.out.print(k); //输出k

System.out.println(k); //输出k并回车

System.out.println(); //输出回车

}

}

}

### 二、STL

vector & iterate

#include<vector>

//定义：

vector<int> s; //定义一个空的vector对象，存储的是int类型的元素

vector<int> s(n); //定义一个含有n个int元素的vector对象

vector<int> s(first,last); //定义一个vector对象，并从由迭代器first和last定义的序列[first,last)中复制初值

vector<int> s[]; //定义二维vector

vector<vector<int> >s;

//vector的基本操作

s[i]; //访问容器中的元素

s.front(); //返回首元素

s.back(); //返回尾元素

s.size(); //返回表长

s.empty(); //表为空返回真

s.push\_back(x); //向表尾插入元素x

s.pop\_back(); //删除表尾元素

s.begin(); //返回指向首元素的随机存取迭代器

s.end(); //返回指向尾元素的下一个位置的随机存取迭代器

s.insert(it,x); //向迭代器it指向的元素前插入新元素x

s.insert(it,n,x); //向迭代器it指向的元素前插入n个x

s.insert(it,first.last); //将由迭代器first和last所指定的序列[first,last)插入到迭代器it指向的元素前

s.erase(it); //删除由迭代器it所指向的元素

s.erase(first,last); //删除由迭代器first和last所指定的序列[first,last)

s.reserve(n); //预分配缓冲空间，是存储空间至少可容纳n个元素

s.resize(n); //改变序列的长度，超出的元素将会被删除，如果序列需要扩展(原空间小于n)，元素默认值将填满扩展出的空间

s.resize(n,val); //改变序列的长度，超出的元素将会被删除，如果序列需要扩展(原空间小于n)，将用val填满扩展出的空间

s.clear(); //删除容器中的所有元素

s.swap(v); //将s和另一个vector对象v进行交换

s.assign(first,last); //将序列替换成由迭代器first和last所指定的序列[first,last)

迭代器iterate用法参见下面例子

例子：

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> s;

for(int i=0;i<10;i++)

s.push\_back(i);

// 输出所有元素

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

// printf("%d ",s[i]);

// 用iterator输出所有元素

vector<int>::iterator it;

for(it=s.begin();it!=s.end();it++)

printf("%d ",\*it);

printf("\n");

return 0;

}

map

map，中文意思就是映射，第二个为关键字，由pair组成的红黑树。map内部自建一颗红黑树(一种非严格意义上的平衡二叉树)，这颗树具有对数据自动排序的功能，所以在map内部所有的数据都是有序的。

#include<map>

map<string,int>M;

常见操作：

M[str1]=1 数组形式插入元素

M.insert(map<string,int>::value\_type(str4,4));用insert函数插入value\_type数据

M.insert(pair<string,int>(str4,4)); 用insert函数插入pair数据

M.size() 容器元素个数

M.begin() 容器开始的迭代器

M.end() 容器结束的迭代器

M.find(str2) 查找元素str2，查找成功返回所在的迭代器，失败返回M.end()

M.clear() 清空map

M.lower\_bound() 返回下界

M.upper\_bound() 返回上界

例子：

#include<map>

#include<string>

#include<iostream>

using namespace std;

int main()

{

//定义map和迭代器

map<string,int>M;

map<string,int>::iterator it;

string str1="abc";

string str2="def";

string str3="ghijklmn";

string str4(str3,5,3);

//往map里插入数据

//用数组方式插入数据

M[str1]=1;

M[str2]=2;

M[str3]=3;

//用insert函数插入value\_type数据

M.insert(map<string,int>::value\_type(str4,4));

//用insert函数插入pair数据

M.insert(pair<string,int>("xyz",5));

//输出map里元素个数

printf("%d\n",M.size());

//顺序输出元素

for(it=M.begin();it!=M.end();it++)

cout<<it->first<<" "<<it->second<<endl;

//逆序输出元素

map<string,int>::reverse\_iterator iter;

for(iter=M.rbegin();iter!=M.rend();iter++)

cout<<iter->first<<" "<<iter->second<<endl;

//查找元素str2，没有则返回M.end()

it=M.find(str2);

if(it == M.end())

printf("No result.\n");

else

{

printf("%d\n",M[str2]);

//删除指定元素

M.erase(it);

}

//清空map

M.clear();

return 0;

}

利用map一对多映射的写法：

int main()

{

map<int,multiset<int> > m;

multiset<int>::iterator it;

for(int i=1;i<=10;i++)

m[1].insert(i);

for(it=m[1].begin();it!=m[1].end();it++)

printf("%d\n",\*it);

m[1].clear();

m.clear();

return 0;

}

map一对两个数映射的写法：

声明：map<pair<int,int>,int>mp;

使用：mp[make\_pair(tmp,pre)]=1;

queue

#include<queue>

1.普通声明

queue<int>Q;

2.结构体声明

struct node{

int x,y;

};

queue<node>Q;

常用函数：

Q.empty()，判断队列是否为空，空返回true

Q.front()，返回队首元素

Q.push()，往队列里加元素

Q.pop()，删除队首元素

Q.size()，返回队列中元素个数

stack

#include<stack>

1.普通声明

stack<int>S;

2.结构体声明

struct node{

int x,y;

};

stack<node>S;

常用函数：

S.empty()，判断队列是否为空，空返回true

S.top()，返回队首元素

S.push()，往队列里加元素

S.pop()，删除队首元素

S.size()，返回队列中元素个数

单调栈：

单调递增或者单调递减的栈。

可以求以某个值为最小值（或最大值），向两边扩展出一段连续区间，这个值在该区间内永远是最小值（或最大值）。入栈的元素确定左区间，出栈的元素确定右区间。区间的左边界是栈顶前一个元素的位置+1；区间的右边界是当前位置-1。

priority\_queue

#include<iostream>

#include<functional>

#include<queue>

#include<vector>

using namespace std;

struct cmp1

{

bool operator () (int &a, int &b)

{

return a > b ; //从小到大排序，值小的优先级别高

}

};

struct cmp2

{

bool operator () (int &a, int &b)

{

return a < b; //从大到小

}

};

struct number1

{

int x;

bool operator < (const number1 &a)const

{

return x > a.x; //从小到大，x小的优先级别高

}

};

struct number2

{

int x;

bool operator < (const number2 &a)const

{

return x < a.x; //从大到小，x大的优先级别高

}

};

int a[] = {14,10,56,7,83,22,36,91,3,47,72,0};

//数组类型是number1和number2

number1 num1[] ={14,10,56,7,83,22,36,91,3,47,72,0};

number2 num2[] ={14,10,56,7,83,22,36,91,3,47,72,0};

int main()

{

//常见的几种用法：

priority\_queue<int>que; //采用默认优先级构造队列从大到小

priority\_queue<int, vector<int>, cmp1 >que1;

priority\_queue<int, vector<int>, cmp2 >que2;

priority\_queue<int, vector<int>, greater<int> > que3; //functional 头文件自带的

priority\_queue<int, vector<int>, less<int> > que4; //functional 头文件自带的

priority\_queue<number1> que5;

priority\_queue<number2> que6;

/\*常用的函数：

\* que.empty()，判断队列是否为空，空返回true

\* que.top()，返回队首元素

\* que.push()，往队列里加元素

\* que.pop()，删除队首元素

\* que.size()，返回队列中元素个数

\*/

return 0;

}

set

一个集合(set)是一个容器，它其中所包含的元素的值是唯一的。集合中的元素按一定的顺序排列，并被作为集合中的实例。

一个集合通过一个链表来组织，在插入操作和删除操作上比向量(vector)快，但查找或添加末尾的元素时会有些慢。

具体实现采用了红黑树的平衡二叉树的数据结构。

#include <set>

set<int> myset;

//常见操作

begin() 返回指向第一个元素的迭代器

clear() 清除所有元素

count() 返回某个值元素的个数

empty() 如果集合为空，返回true(真）

end() 返回指向最后一个元素之后的迭代器，不是最后一个元素

equal\_range() 返回集合中与给定值相等的上下限的两个迭代器

erase() 删除集合中的元素

find() 返回一个指向被查找到元素的迭代器

get\_allocator() 返回集合的分配器

insert() 在集合中插入元素

lower\_bound() 返回指向小于（或等于）某值的第一个元素的迭代器

upper\_bound() 返回指向大于某值的第一个元素的迭代器

key\_comp() 返回一个用于元素间值比较的函数

max\_size() 返回集合能容纳的元素的最大限值

rbegin() 返回指向集合中最后一个元素的反向迭代器

rend() 返回指向集合中第一个元素的反向迭代器

size() 集合中元素的数目

swap() 交换两个集合变量

upper\_bound() 返回大于某个值元素的迭代器

value\_comp() 返回一个用于比较元素间的值的函数

例子：

#include <iostream>

#include <set>

using namespace std;

int main()

{

//定义容器set和迭代器

set<int> myset;

set<int>::iterator it,itlow,itup;

//往容器里插入元素

for(int i=1;i<=10;i++)

myset.insert(i\*10);

myset.insert(55);

//输出容器元素个数

printf("%d\n",(int) myset.size());

//11

//依次输出容器元素

for(it=myset.begin();it!=myset.end();it++)

printf("%d ",\*it);

printf("\n");

//10 20 30 40 50 55 60 70 80 90 100

//依次输出容器元素并删除

// while(!myset.empty())

// {

// printf("%d ",\*myset.begin());

// myset.erase(myset.begin());

// }

// printf("\n");

//从容器里查找元素70并删除

it=myset.find(70);

myset.erase(it);

//输出容器里20和25的个数

printf("%d\n",myset.count(20));

printf("%d\n",myset.count(25));

//1

//0

//找到第一个小于等于30的容器，第一个大于60的容器

itlow=myset.lower\_bound(30);

itup=myset.upper\_bound(60);

for(it=itlow;it!=itup;it++)

printf("%d ",\*it);

printf("\n");

//40 50 55 60

//找到集合中给定值相等的上下限的两个迭代器

pair<set<int>::iterator, set<int>::iterator > ret;

ret=myset.equal\_range(30);

printf("lower bound points to: %d\n",\*ret.first);

printf("upper bound points to: %d\n",\*ret.second);

//lower bound points to: 30

//upper bound points to: 40

//清空容器

myset.clear();

return 0;

}

//自定义排序方法的set定义方法：

struct Node{

int x,y;

};

struct cmp{

bool operator ()(const Node& a,const Node& b) //x小的优先，x相同时y小的优先

{

if(a.x == b.x)

return a.y < b.y;

return a.x < b.x;

}

};

set<Node,cmp> myset;

list

list就是一双向链表，可高效地进行插入删除元素。包括构造、方法等。

常用函数：

front()返回第一个元素的引用

back()返回最后一元素的引用

begin()返回第一个元素的容器

end()返回最后一个元素的下一位置的容器(list为空时end()=begin())

rbegin()返回链表最后一元素的后向容器

rend()返回链表第一元素的下一位置的后向容器

push\_back()增加一元素到链表尾

push\_front()增加一元素到链表头

pop\_back()删除链表尾的一个元素

pop\_front()删除链表头的一元素

clear()删除所有元素

erase()删除一个元素或一个区域的元素(两个重载函数)

remove()删除链表中匹配值的元素(匹配元素全部删除)

empty()判断是否链表为空

max\_size()返回链表最大可能长度

size()返回链表中元素个数

reverse()反转链表

sort()对链表排序，默认升序(可自定义回调函数)

merge()合并两个有序链表并使之有序

insert()在指定位置插入一个或多个元素(三个重载函数)

swap()交换两个链表(两个重载)

unique()删除相邻重复元素

例子：

#include<iostream>

#include<list>

using namespace std;

int main()

{

//定义链表

list<int> list1;

//从链表尾部加入元素

for(int i=6;i<=10;i++)

list1.push\_back(i);

//从链表头部加入元素

for(int i=5;i>=1;i--)

list1.push\_front(i);

//list2是list1的copy

list<int> list2(list1);

//获取链表元素个数

printf("%d\n",list1.size());

//顺序输出链表各个元素

list<int>::iterator it1;

for(it1=list1.begin();it1!=list1.end();it1++)

printf("%d ",\*it1);

printf("\n");

//分别从头部和尾部各删除一个元素

list1.pop\_front();

list1.pop\_back();

//删除第1个元素

list1.erase(list1.begin());

//删除链表中匹配值的元素

list1.remove(8);

//往链表里插入元素的三种方法

list1.insert(++list1.begin(),20); //在第1个数后插入元素20

list1.insert(list1.begin(),5,30); //在最前面插入5个元素，值均为30

list1.insert(++list1.begin(),list2.begin(),list2.end()); //把list2从头到尾插入到list1第1个数后面

//逆序输出链表

list<int>::reverse\_iterator it2;

for(it2=list1.rbegin();it2!=list1.rend();it2++)

printf("%d ",\*it2);

printf("\n");

//链表反转

list2.reverse();

//链表排序，默认升序

list1.sort();

list1.sort(greater<int>()); //降序

//链表的交换

list1.swap(list2);

//删除相邻重复的元素

list1.unique();

//把list2合并到list1两个链表，并使之有序，默认升序

list1.merge(list2); //合并后list2为空

list1.merge(list2,greater<int>()); //降序，合并后list2为空

//清除list中的所有元素

list1.clear();

list2.clear();

return 0;

}

例2：

list中自定义sort函数写法：

#define N 1010

struct Node{

int num;

}node[N];

struct S{

bool operator()(const Node& t1,const Node& t2) //自定义排序函数

{

if(t1.num < t2.num) //升序，‘>’降序

return true;

return false;

}

};

int main()

{

int f[6]={2,3,4,5,1};

list<Node> list1;

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

{

node[i].num=f[i];

list1.push\_back(node[i]);

}

list1.sort(S());

list<Node>::iterator ite;

for(ite=list1.begin();ite!=list1.end();ite++)

printf("%d\n",ite->num);

return 0;

}

string

常用操作：

int size()const; //返回当前字符串的大小

int length()const; //返回当前字符串的长度

bool empty()const; //当前字符串是否为空

bool operator==(const string &s1,const string &s2)const;//比较两个字符串是否相等，运算符">","<",">=","<=","!="均被重载用于字符串的比较；

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

void swap(string &s2); //交换当前字符串与s2的值

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

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

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

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

//查找成功时返回所在位置，失败返回string::npos的值

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

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

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

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

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

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

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

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

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

//前4个函数在p0位置插入字符串s中pos开始的前n个字符

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

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

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

next\_permutation

全排列函数

next\_permutation(start,end);

返回值是真或假

#include<algorithm>

#include<iostream>

using namespace std;

int main()

{

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

sort(a,a+4);

while(next\_permutation(a,a+4))

{

for(int i=0;i<4;i++)

printf("%d ",a[i]);

printf("\n");

}

return 0;

}

Others

binary\_search(first,last,value);

在指定的区间里二分查找元素value，如果找到返回true，否则返回false；

int main()

{

int f[]={1,2,3,5,6,7,8};

if(binary\_search(f,f+4,8))

printf("yes");

else

printf("no");

return 0;

}

//输出no，如果改为2，则输出yes。

swap(a,b) //交换两个元素

reverse(first,last) //反转指定的区间元素。

int main()

{

int f[]={1,2,3,4,5,6,7};

reverse(f,f+3);

for(int i=0;i<7;i++)

printf("%d ",f[i]);

return 0;

}

//输出3 2 1 4 5 6 7

lower\_bound(first,last,tar);

返回有序数组中大于或等于tar的第一个元素位置

int main()

{

int f[]={1,2,4,5,6,7};

int pos=lower\_bound(f,f+6,3)-f;

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

return 0;

}

//输出2

### 三、组合数学

Lucas定理

/\*

求C(n,m)%p, p为素数，适用于n和m很大的时候

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

把n写成p进制a[n]a[n-1]a[n-2]...a[0]，把m写成p进制b[n]b[n-1]b[n-2]...b[0]， 则C(n,m)与C(a[n],b[n])\*C(a[n-1],b[n-1])\*C(a[n-2],b[-2])\*....C(a[0],b[0])模p同余。

\*注：p不能大于10^5，不大于情况下用逆元的方法可以解决，如果大了就不能解决。

\*/

#define LL \_\_int64

#define N 10001000

#define M 1000000007ll

LL fac[N];

void init(LL p)

{

LL i;

fac[0] =1;

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

fac[i] = fac[i-1]\*i % p;

}

LL exp\_mod(LL a, LL b, LL p)

{

LL tmp = a % p, ans =1;

while(b)

{

if(b & 1) ans = ans \* tmp % p;

tmp = tmp\*tmp % p;

b >>=1;

}

return ans;

}

LL C(LL n, LL m, LL p)

{

if(m > n)

return 0;

return fac[n]\*exp\_mod(fac[m]\*fac[n-m], p-2, p) % p;//逆元

}

LL Lucas(LL n, LL m, LL p)

{

if(m ==0)

return 1;

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

}

int main()

{

int T;

LL n,m,p;

scanf("%d",&T);

while(T--)

{

scanf("%I64d %I64d %I64d",&n,&m,&p);

init(p);

printf("%I64d\n",Lucas(n,m,p));

}

return 0;

}

互质数最大不能组合数

给出两个互质数n和m，求最大不能组合数和不能组合数的个数

结论：最大不能组合数：(m-1)\*n-m; 不能组合数的个数：(n-1)\*(m-1)/2.

证明：

任何数分成m个剩余类，分别为 mk，mk+1，mk+2，……，mk+(m-1) 分别记为{0(mod m)}，{1(mod m)}……

而n的倍数肯定分布在这m个剩余类中，因为Gcd(m，n)=1，所以每个剩余类中都有一些数是n的倍数，并且是平均分配它的旁证，

设 kmin = min{ k | nk ∈ {i (mod m)} }, i ∈ [0, m) 则 nkmin 是{i (mod m)}中n的最小倍数。

特别的，nm ∈ {0 (mod m)} nkmin 是个标志，它表明{i (mod m)}中nkmin 后面所有数，即nkmin + jm必定都能被组合出来。

那也说明最大不能组合数必定小于nkmin，我们开始寻找max{ nkmin }。

Lcm(m, n) = mn，所以很明显(m-1)n是最大的，因为(m-1)n是nkmin 中的最大值，所以在剩下的m-1个剩余类中，

必定有比它小并且能被m和n组合，这些数就是(m-1)n -1，(m-1)n -2，……，(m-1)n -(m-1)

所以最大不能被组合数就是(m-1)n-m。

如果m和n不互素，那{1 (mod m)}不能被m组合，同样也不能被n和m组合。我们能求出各个剩余类的nkmin之后，不能组合数的个数就是每个剩余类中小于各自nkmin的数的个数总和。

观察如下：

M = 5，N = 3

{0(mod 5)}：0，5，10，15……

{1(mod 5)}：1，6，11，16……

{2(mod 5)}：2，7，12，17……

{3(mod 5)}：3，8，13，18……

{4(mod 5)}：4，9，14，19……

红色的就是不能组合数，可以看出在剩余类中它的数目有规律Total = [0+1+2] + [0+1]

因为m和n互质，必有一个不完全周期。

整理以后，可得公式 Total = (n-1)\*(m-1)/2

母函数

母函数的思想就是使用幂级数来表示一个离散数列。例如对于一个数列是a={a0,a1,a2,-------ak}，那么我们可以母函数的定义就是G(x)=a0+a1\*x+a2\*x^2+a3\*x^3------an\*x^n，这里的G(x)就是数列a的母函数。

母函数可以用来解决整数拆分问题，下面举一个例子来说明。

例如我们有重量为1克、2克、3克、4克的砝码各一个，那么问你这些砝码可以称出几中重量？每一种重量有多少种方案数？

对于这个问题我们可以这样定义母函数G(x)=(1+x)\*(1+x^2)\*(1+x^3)\*(1+x^4)，那么这个时候我们把G(x)展开之后得到一个多项式，

(1+x)(1+x2)(1+x3)(1+x4)=(1+x+x2+x3)(1+x3+x4+x7)

=1+x+x2+2x3+2x4+2x5+2x6+2x7+x8+x9+x10

从上面的函数知道：可称出从1克到10克，系数便是方案数。

例如右端有2x5项，即称出5克的方案有2：5=3+2=4+1；同样，6=1+2+3=4+2；10=1+2+3+4。

故称出6克的方案有2，称出10克的方案有1。

下面我们说说HDU 1284 的“钱币兑换问题”吧，题目的意思就是说给你三种面值的币分别是1分、2分、3分，那么问你一个钱数n有几种方案可以拼凑得到这个钱数n？

这道题还是比较好做的，我们根据母函数的定义，以及多项式的每一项和系数所表示的含义，我们可以定义母函数G(x)=(1+x+x^2+x^3+……)\*(1+x^2+x^4+x^6+x^8+……)\*(1+x^3+x^6+x^9+……)，那么我们使用母函数的额展开式对应的指数就是能表示的钱币数，系数就是表示该钱币数的方案数。

所以问题就是求出对应的n的系数就可以了，就是模拟手工多项式的展开。

#include <cstring>

#include<iostream>

using namespace std;

const int Max=32769;

int ans[Max];

int tans[Max];

int main()

{

int i,j,n;

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

ans[i]=1;

memset(tans,0,sizeof(tans));

for(int k=2;k<=3;k++)

{

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

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

tans[i+j]+=ans[i];

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

{

ans[i]=tans[i];

tans[i]=0;

}

}

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

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

return 0;

}

容斥原理

int plen,p[N];

//分解质因子

void Getfac(int n)

{

plen=0;

for(int i=0;prime[i]\*prime[i] <= n ;i++)

{

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

{

p[plen++]=prime[i];

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

n/=prime[i];

}

if(n == 1)

break;

}

if(n > 1)

p[plen++]=n;

}

//容斥原理求解区间[1,n]内与m不互质个数

int dfs(int tmp,int n)

{

int s=0;

for(int i=tmp;i<plen;i++)

s=s+n/p[i]-dfs(i+1,n/p[i]);

return s;

}

//容斥原理求解区间[1,n]内与m不互质数之和(tmp表示当前位，num表示当前因子，c表示当前因子数，n表示区间最大值，plen和p[]见函数分解质因子)

int ans;

void dfs(int tmp,int num,int c,int n)

{

int t,tt;

for(int i=tmp;i<plen;i++)

{

if(num\*p[i] <= n)

{

tt=num\*p[i];

t=n/tt;

t=tt\*t+t\*(t-1)/2\*tt;

if((c+1) & 1)

ans+=t;

else

ans-=t;

dfs(i+1,num\*p[i],c+1,n);

}

}

}

void work(int n,int m)

{

Getfac(m);

int res=dfs(0,n);

//res为区间[1,n]内与m不互质个数，n-res为互质个数

ans=0;

dfs(0,1,0,n);

//ans为区间[1,n]内与m不互质数之和

//所有互质数和=sigma(1,n)-ans

}

整数划分

#include <stdio.h>

int ans,a[100] = { 0 };

void shuchu(int m) {

int i;

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

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

printf("+%d", a[i]);

printf("\n");

}

//递归打印每个数

void fenjie(int n, int m) {

int i;

if (n == 0)

{

ans++;

shuchu(m);

}

else

for (i = n; i >= 1; i--)

if (m == 0 || i <= a[m - 1]) {

a[m] = i;

fenjie(n - i, m + 1);

}

}

int main()

{

int n, m = 0;

//printf("please input a number(0<n<100): ");

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

{

ans=0;

fenjie(n, m);

printf("total num: %d\n\n",ans);

}

return 0;

}

//递归求总个数

int q(int n,int m)

{

if(n<1||m<1)

return 0;

if(n==1||m==1)

return 1;

if(n<m)

return q(n,n);

if(n==m)

return q(n,m-1)+1;

return q(n,m-1)+q(n-m,m);

}

通项公式：

p(k)=sigma(p(k-i\*(3\*i-1)/2)+p(k-i\*(3\*i+1)/2))\*(-1)^(i+1);

p(k)=p(k-1)+p(k-2)-p(k-5)-p(k-7)+p(k-12)+p(k-15)-p(k-22)-p(k-26)+……

+(1, 2), -(5,7), +(12,15), -(22,26)……

1，5，12，22满足公式n\*(3\*n-1)/2。

由五边形原理推得：

五边形定理，分割函数，把数n拆成几个数（小于等于n）相加的形式拆法

(1-x)(1-x^2)(1-x^3).... = 1-x-x^2+x^5+x^7-x^12-x^15+x^22+x^26+......

维基上有详细说明http://en.wikipedia.org/wiki/Partition\_(number\_theory)

把数n拆成几个数相加的形式，一共有2^(n-1)种

如果n=4，有

4=1+1+1+1

4=1+1+2

4=1+2+1

4=2+1+1

4=1+3

4=2+2

4=3+1

4=4

组合数打表

const int MN=505,MV=15005;

LL c[MV][MN];

void initcom() //c(n, i) == c[n][i]

{

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

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

c[i][0]=1;

c[1][1]=1;

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

{

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

{

if(j == i)

{

c[i][j]=1;

break;

}

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

}

}

}

组合数普通求法

//适用于n和r都比较小的情况

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

{

if(n-r > r)

r=n-r;

int s=1;

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

{

s\*=(n-i);

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

s/=j;

}

return s;

}

### 四、数论

Baby Step Giant Step(n不为素数)

求解模方程a^x=b(mod n)，n不为素数。拓展Baby Step Giant Step

//拓展欧几里得定理，求ax+by=gcd(a,b)的一组解(x,y),d=gcd(a,b)

void gcd\_mod(LL a,LL b,LL &d,LL &x,LL &y)

{

if(!b)

{

d=a;

x=1;

y=0;

}

else

{

gcd\_mod(b,a%b,d,y,x);

y-=x\*(a/b);

}

}//求解模方程d\*a^(x-c)=b(mod n)。d,a和n互质，无解返回-1

LL Log\_mod (LL a,LL b,LL n,LL c,LL d)

{

LL m,e=1,i,x,y,dd;

m=(LL)ceil(sqrt(n+0.5)); //x=i\*m+j

map<LL,LL>f;

f[1]=m;

for(i=1;i<m;i++) //建哈希表，保存a^0,a^1,...,a^m-1

{

e=(e\*a)%n;

if(!f[e])

f[e]=i;

}

e=(e\*a)%n;//e=a^m

for(i=0;i<m;i++)//每次增加m次方，遍历所有1<=f<=n

{

gcd\_mod(d,n,dd,x,y);

//d\*x+n\*y=1-->(d\*x)%n=1-->d\*(x\*b)%n==b

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

if(f[x])

{

LL num=f[x];

f.clear();//需要清空，不然会爆内存

return c+i\*m+(num==m?0:num);

}

d=(d\*e)%n;

}

return -1;

}

LL solve(LL a,LL b,LL n)

{

if(b >= n) return -1;

if(b == 0) return 0;

LL ans=0,c=0,d=1,t;

while((t=gcd(a,n))!=1)

{

if(b%t)

return -1;

c++;

n/=t;b/=t;

d=d\*a/t%n;

if(d == b) //特判下是否成立

{

ans=c;

break;

}

}

if(ans != 0)

return ans;

return Log\_mod(a,b,n,c,d);

}

/\*

初始d=1,c=0,i=0;

1.令g=gcd(a,n),若g==1则执行下一步。否则由于a^x=k\*n+b;(k为某一整数),则(a/g)\*a^k=k\*(n/g)+b/g,(b/g为整除，若不成立则无解)

令n=n/g，d=d\*a/g，b=b/g,c++则d\*a^(x-c)=b(mod n),接着重复1步骤。

2.通过1步骤后，保证了a和d都与n互质，方程转换为d\*a^(x-c)=b(mod n)。由于a和n互质，所以由欧拉定理a^phi(n)==1(mod n),(a,n互质)

可知，phi(n)<=n,a^0==1(mod n),所以构成循环，且循环节不大于n。从而推出如果存在解，则必定1<=x<n。(在此基础上我们就可以用

Baby Step Giant Step方法了)

3.令m=ceil(sqrt(n)),则m\*m>=n。用哈希表存储a^0,a^1,..,a^(m-1)，接着判断1~m\*m-1中是否存在解。

4.为了减少时间，所以用哈希表缩减复杂度。分成m次遍历，每次遍历a^m长度。由于a和d都与n互质，所以gcd(d,n)=1，

所以用拓展的欧几里德定理求得d\*x+n\*y=gcd(d,n)=1,的一组整数解(x,y)。则d\*x+n\*y=1-->d\*x%n=(d\*x+n\*y)%n=1-->d\*(x\*b)%n=((d\*x)%n\*b%n)%n=b。

所以若x\*b在哈希表中存在，值为k，则a^k\*d=b(mod n),答案就是ans=k+c+i\*m。如果不存在，则令d=d\*a^m,i++后遍历下一个a^m，直到遍历a^0到a^(m-1)还未找到，则说明不解并退出。

\*/

Baby Step Giant Step(n为素数)

求解模方程a^x=b(mod p)。p为素数，无解返回-1

//快速幂求a^b

LL Mod\_exp(LL a,LL b,LL c)

{

LL ans=1;

while(b)

{

if(b & 1)

ans=ans\*a%c;

a=(a%c)\*(a%c)%c;

b/=2;

}

return ans;

}

//费马小定理a^(p-1)=1(mod p),p为素数。a^0=1,所以循环节小于等于p，即如果存在解，则最小解x<=p

LL Log\_mod (LL a,LL b,LL p)

{

if(a%p == 0) return -1;

LL m,v,e=1;

m=(LL)ceil(sqrt(p+0.5)); //x=i\*m+j

//v=inv(pow\_mod(a,m,n),n); //a^m\*v=1(mod n)

v=Mod\_exp(a,p-m-1,p);

map<LL,LL>x;

x[1]=m;

for(LL i=1;i<m;i++) //建哈希表，保存x^0,x^1,...,x^m-1

{

e=(e\*a)%p;

if(!x[e])

x[e]=i;

}

for(LL i=0;i<m;i++)//每次增加m次方，遍历所有1<=x<=n

{

if(x[b])

{

LL num=x[b];

x.clear();//需要清空，不然会爆内存

return i\*m+(num==m?0:num);

}

b=(b\*v)%p; //b=b/(a^m)

}

return -1;

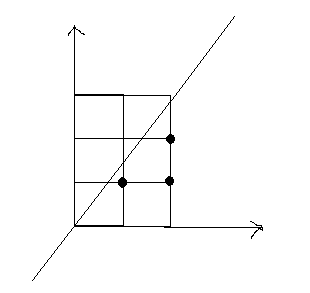
}

lattice-count栅格计数问题

已知n,m>0, a,b>=0, 求下列公式结果：

该式子解决了直线m\*y=a+b\*x在x从0到n的范围内有多少个整数点（不包括x轴的点）的问题。

例如直线为5\*y=7\*x，n为3，结果为3，如下图：



LL lattice\_count(LL n,LL a,LL b,LL m)

{

if(b == 0)

return n\*(a/m);

if(a >= m)

return n\*(a/m)+lattice\_count(n,a%m,b,m);

if(b >= m)

return (n-1)\*n/2\*(b/m)+lattice\_count(n,a,b%m,m);

return lattice\_count((a+b\*n)/m,(a+b\*n)%m,m,b);

}

Stirling数应用

Bell数

又称贝尔数，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)!]

第一类Stirling数

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)+n\*S(n,k).

第二类Stirling数

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)+k\*S(n-1,k).

Bell数和Stirling数的关系为：

每个Bell数都是第二类Stirling数的和

B(n)=Sum(1,n) S(n,k).

//打印前50个贝尔数以及第二类Stirling数

int bell[N],num[N][N];

void init()

{

num[0][0]=1;

bell[0]=bell[1]=1;

for(int i=1;i<=50;i++)

{

num[i][1]=1;

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

num[i][j]=(num[i-1][j-1]+j\*num[i-1][j])%MOD;

bell[i]=0;

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

bell[i]=(bell[i]+num[i][k])%MOD;

}

}

乘法逆元

/\*

\* 当要求(a/b)%p的值,由于除法没有同余公式,要用乘法逆元改成可同余的相乘

\* 逆元满足 k\*a = 1 (mod p)

\* b\*k=p\*x+1,把k带入(a\*k)%p就得到(a/b)%p

\* 扩展的欧几里德算法求乘法逆元模板

\* (a/b) mod p = ( a\*[b与p(互质)乘法逆元]k ) mod p

\*/

#define LL long long

LL exGcd(LL a,LL b,LL &x,LL &y)

{

if(b==0)

{

x=1; y=0;

return a;

}

LL temp=exGcd(b,a%b,x,y);

LL t=x;

x=y;

y=t-a/b\*y;

return temp;

}

//求出a的逆元

LL cal(LL a,LL p)

{

LL x,y;

exGcd(a,p,x,y);

x=(x%p+p)%p;

return x;

}

/\*

\* 模板2，a的逆元是a^(p-2)，条件：p是质数，gcd(a,p)==1

\* 由费马小定理推导:

\* a^(p-1) ≡ 1(mod p) ==> a\*a^(p-2) ≡ 1(mod p) ==> a的逆元是a^(p-2)

\*/

LL mod\_exp(LL a,LL b,LL c)

{

LL ans=1;

while(b)

{

if(b & 1)

ans=ans\*a%c;

a=a\*a%c;

b/=2;

}

return ans;

}

//求a的逆元

LL inv( LL a ,LL p)

{

return mod\_exp(a, p - 2 )%p;

}

大数模板

#define MAXN 9999

#define MAXSIZE 10

#define DLEN 4

class BigNum

{

private:

int a[500]; //a数组一位记录一个四位的整数，控制a数组大小=结果总长/4

int len; //大数长度

public:

BigNum(){ len = 1;memset(a,0,sizeof(a)); } //构造函数

BigNum(const int); //将一个int类型的变量转化为大数

BigNum(const char\*); //将一个字符串类型的变量转化为大数

BigNum(const BigNum &); //拷贝构造函数

BigNum &operator=(const BigNum &); //重载赋值运算符，大数之间进行赋值运算

friend istream& operator>>(istream&, BigNum&); //重载输入运算符

friend ostream& operator<<(ostream&, BigNum&); //重载输出运算符

BigNum operator+(const BigNum &) const; //重载加法运算符，两个大数之间的相加运算

BigNum operator-(const BigNum &) const; //重载减法运算符，两个大数之间的相减运算

BigNum operator\*(const BigNum &) const; //重载乘法运算符，两个大数之间的相乘运算

BigNum operator/(const int &) const; //重载除法运算符，大数对一个整数进行相除运算

BigNum operator^(const int &) const; //大数的n次方运算

int operator%(const int &) const; //大数对一个int类型的变量进行取模运算

bool operator>(const BigNum & T)const; //大数和另一个大数的大小比较

bool operator>(const int & t)const; //大数和一个int类型的变量的大小比较

void print(); //输出大数

};

BigNum::BigNum(const int b) //将一个int类型的变量转化为大数

{

int c,d = b;

len = 0;

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

while(d > MAXN)

{

c = d - (d / (MAXN + 1)) \* (MAXN + 1);

d = d / (MAXN + 1);

a[len++] = c;

}

a[len++] = d;

}

BigNum::BigNum(const char\*s) //将一个字符串类型的变量转化为大数

{

int t,k,index,l,i;

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

l=strlen(s);

len=l/DLEN;

if(l%DLEN)

len++;

index=0;

for(i=l-1;i>=0;i-=DLEN)

{

t=0;

k=i-DLEN+1;

if(k<0)

k=0;

for(int j=k;j<=i;j++)

t=t\*10+s[j]-'0';

a[index++]=t;

}

}

BigNum::BigNum(const BigNum & T) : len(T.len) //拷贝构造函数

{

int i;

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

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

a[i] = T.a[i];

}

BigNum & BigNum::operator=(const BigNum & n) //重载赋值运算符，大数之间进行赋值运算

{

int i;

len = n.len;

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

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

a[i] = n.a[i];

return \*this;

}

istream& operator>>(istream & in, BigNum & b) //重载输入运算符

{

char ch[MAXSIZE\*4];

int i = -1;

in>>ch;

int l=strlen(ch);

int count=0,sum=0;

for(i=l-1;i>=0;)

{

sum = 0;

int t=1;

for(int j=0;j<4&&i>=0;j++,i--,t\*=10)

{

sum+=(ch[i]-'0')\*t;

}

b.a[count]=sum;

count++;

}

b.len =count++;

return in;

}

ostream& operator<<(ostream& out, BigNum& b) //重载输出运算符

{

int i;

cout << b.a[b.len - 1];

for(i = b.len - 2 ; i >= 0 ; i--)

{

cout.width(DLEN);

cout.fill('0');

cout << b.a[i];

}

return out;

}

BigNum BigNum::operator+(const BigNum & T) const //两个大数之间的相加运算

{

BigNum t(\*this);

int i,big; //位数

big = T.len > len ? T.len : len;

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

{

t.a[i] +=T.a[i];

if(t.a[i] > MAXN)

{

t.a[i + 1]++;

t.a[i] -=MAXN+1;

}

}

if(t.a[big] != 0)

t.len = big + 1;

else

t.len = big;

return t;

}

BigNum BigNum::operator-(const BigNum & T) const //两个大数之间的相减运算

{

int i,j,big;

bool flag;

BigNum t1,t2;

if(\*this>T)

{

t1=\*this;

t2=T;

flag=0;

}

else

{

t1=T;

t2=\*this;

flag=1;

}

big=t1.len;

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

{

if(t1.a[i] < t2.a[i])

{

j = i + 1;

while(t1.a[j] == 0)

j++;

t1.a[j--]--;

while(j > i)

t1.a[j--] += MAXN;

t1.a[i] += MAXN + 1 - t2.a[i];

}

else

t1.a[i] -= t2.a[i];

}

t1.len = big;

while(t1.a[len - 1] == 0 && t1.len > 1)

{

t1.len--;

big--;

}

if(flag)

t1.a[big-1]=0-t1.a[big-1];

return t1;

}

BigNum BigNum::operator\*(const BigNum & T) const //两个大数之间的相乘运算

{

BigNum ret;

int i,j,up;

int temp,temp1;

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

{

up = 0;

for(j = 0 ; j < T.len ; j++)

{

temp = a[i] \* T.a[j] + ret.a[i + j] + up;

if(temp > MAXN)

{

temp1 = temp - temp / (MAXN + 1) \* (MAXN + 1);

up = temp / (MAXN + 1);

ret.a[i + j] = temp1;

}

else

{

up = 0;

ret.a[i + j] = temp;

}

}

if(up != 0)

ret.a[i + j] = up;

}

ret.len = i + j;

while(ret.a[ret.len - 1] == 0 && ret.len > 1)

ret.len--;

return ret;

}

BigNum BigNum::operator/(const int & b) const //大数对一个整数进行相除运算

{

BigNum ret;

int i,down = 0;

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

{

ret.a[i] = (a[i] + down \* (MAXN + 1)) / b;

down = a[i] + down \* (MAXN + 1) - ret.a[i] \* b;

}

ret.len = len;

while(ret.a[ret.len - 1] == 0 && ret.len > 1)

ret.len--;

return ret;

}

int BigNum::operator %(const int & b) const //大数对一个int类型的变量进行取模运算

{

int i,d=0;

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

{

d = ((d \* (MAXN+1))% b + a[i])% b;

}

return d;

}

BigNum BigNum::operator^(const int & n) const //大数的n次方运算

{

BigNum t,ret(1);

int i;

if(n<0)

exit(-1);

if(n==0)

return 1;

if(n==1)

return \*this;

int m=n;

while(m>1)

{

t=\*this;

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

{

t=t\*t;

}

m-=i;

ret=ret\*t;

if(m==1)

ret=ret\*(\*this);

}

return ret;

}

bool BigNum::operator>(const BigNum & T) const //大数和另一个大数的大小比较

{

int ln;

if(len > T.len)

return true;

else if(len == T.len)

{

ln = len - 1;

while(a[ln] == T.a[ln] && ln >= 0)

ln--;

if(ln >= 0 && a[ln] > T.a[ln])

return true;

else

return false;

}

else

return false;

}

bool BigNum::operator >(const int & t) const //大数和一个int类型的变量的大小比较

{

BigNum b(t);

return \*this>b;

}

void BigNum::print() //输出大数

{

int i;

printf("%d",a[len-1]);

for(i = len - 2 ; i >= 0 ; i--)

printf("%04d",a[i]);

printf("\n");

}

int main()

{

BigNum a,b;

int c;

cin>>a>>b>>c;

(a+b).print();

(a-b).print();

(a\*b).print();

(a/c).print();

return 0;

}

大素数判断(miller-rabin算法)和质因数分解(Pollard\_rho算法)

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

// Miller\_Rabin 算法进行素数测试

//速度快，而且可以判断 <2^63的数

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

const int S=20;//随机算法判定次数，S越大，判错概率越小

//计算 (a\*b)%c. a,b都是LL的数，直接相乘可能溢出的

// a,b,c <2^63

LL mult\_mod(LL a,LL b,LL c)

{

a%=c;

b%=c;

LL ret=0;

while(b)

{

if(b&1){ret+=a;ret%=c;}

a<<=1;

if(a>=c)a%=c;

b>>=1;

}

return ret;

}

//计算 x^n %c

LL pow\_mod(LL x,LL n,LL mod)//x^n%c

{

if(n==1)return x%mod;

x%=mod;

LL tmp=x;

LL ret=1;

while(n)

{

if(n&1) ret=mult\_mod(ret,tmp,mod);

tmp=mult\_mod(tmp,tmp,mod);

n>>=1;

}

return ret;

}

//以a为基,n-1=x\*2^t a^(n-1)=1(mod n) 验证n是不是合数

//一定是合数返回true,不一定返回false

bool check(LL a,LL n,LL x,LL t)

{

LL ret=pow\_mod(a,x,n);

LL last=ret;

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

{

ret=mult\_mod(ret,ret,n);

if(ret==1&&last!=1&&last!=n-1) return true;//合数

last=ret;

}

if(ret!=1) return true;

return false;

}

// Miller\_Rabin()算法素数判定

//是素数返回true.(可能是伪素数，但概率极小)

//合数返回false;

bool Miller\_Rabin(LL n)

{

if(n<2)return false;

if(n==2)return true;

if((n&1)==0) return false;//偶数

LL x=n-1;

LL t=0;

while((x&1)==0){x>>=1;t++;}

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

{

LL a=rand()%(n-1)+1;//rand()需要stdlib.h头文件

if(check(a,n,x,t))

return false;//合数

}

return true;

}

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

//pollard\_rho 算法进行质因数分解

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

LL factor[100];//质因数分解结果（刚返回时是无序的）

int tol;//质因数的个数。数组小标从0开始

LL gcd(LL a,LL b)

{

if(a==0)return 1;//???????

if(a<0) return gcd(-a,b);

while(b)

{

LL t=a%b;

a=b;

b=t;

}

return a;

}

LL Pollard\_rho(LL x,LL c)

{

LL i=1,k=2;

LL x0=rand()%x;

LL y=x0;

while(1)

{

i++;

x0=(mult\_mod(x0,x0,x)+c)%x;

LL d=gcd(y-x0,x);

if(d!=1&&d!=x) return d;

if(y==x0) return x;

if(i==k){y=x0;k+=k;}

}

}

//对n进行素因子分解

void findfac(LL n)

{

if(Miller\_Rabin(n))//素数

{

factor[tol++]=n;

return;

}

LL p=n;

while(p>=n)p=Pollard\_rho(p,rand()%(n-1)+1);

findfac(p);

findfac(n/p);

}

int main()

{

//srand(time(NULL));//需要time.h头文件//POJ上G++不能加这句话

LL n;

while(scanf("%I64d",&n)!=EOF)

{

tol=0;

findfac(n);

for(int i=0;i<tol;i++)printf("%I64d ",factor[i]);

printf("\n");

if(Miller\_Rabin(n))printf("Yes\n");

else printf("No\n");

}

return 0;

}

多维曼哈顿距离

对于平面上两点(x1,y1),(x2,y2)的曼哈顿距离为|x1-x2|+|y1-y2|

去掉绝对值，将同一点的坐标整理一边可以得到下面四种情况:

1.x1-x2+y1-y2->(x1+y1)-(x2+y2)

2.x2-x1+y1-y2->(-x1+y1)-(-x2+y2)

3.x1-x2+y2-y1->(x1-y1)-(x2-y2)

4.x2-x1+y2-y1->(-x1-y1)-(-x2-y2)

可以发现左半部分和右半部分的符号完全相同，我们假设0代表’+’,1代表’-‘,则有00 01 10 11.

我们可以把所有的点在k维空间下的2^k种状态处理出。最大距离肯定在2^k种状态下某两个点的状态之差取最大值。

静态(poj 2926)：

struct Point{

double x[6];

}point[N];

#define N 100010

int n,tot,dim=5;

double minn[35],maxn[35];

void solve()

{

tot=1<<dim;

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

{

minn[i]=INF;

maxn[i]=-INF;

}

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

{

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

{

double res=0;

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

{

if(i & (1<<j))

res+=point[k].x[j];

else

res-=point[k].x[j];

}

maxn[i]=Max(maxn[i],res);

minn[i]=Min(minn[i],res);

}

}

double ans=-1;

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

ans=Max(ans,maxn[i]-minn[i]);

printf("%.2lf\n",ans);

}

int main()

{

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

{

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

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

scanf("%lf",&point[i].x[j]);

solve();

}

return 0;

}

动态(hdu 4666)：

用multiset动态维护2^k种状态对应每种状态的所有值，再用结构体数组记录每个点的2^k种状态。

#define N 60010

struct Point{

int x[6],p[40];

}point[N];

int n,dim,tot;

multiset<int> s[40];

void Update(int tmp)

{

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

{

int ans=0;

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

{

if(i & (1<<j))

ans+=point[tmp].x[j];

else

ans-=point[tmp].x[j];

}

point[tmp].p[i]=ans;

}

}

void Insert(int tmp)

{

Update(tmp);

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

s[i].insert(point[tmp].p[i]);

}

void Delete(int tmp)

{

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

s[i].erase(s[i].find(point[tmp].p[i]));

}

void Query()

{

if(s[0].size() < 2)

{

printf("0\n");

return ;

}

int ans=-1;

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

{

int t1=\*(s[i].begin());

int t2=\*(s[i].rbegin());

ans=max(ans,t2-t1);

}

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

}

void solve()

{

tot=1<<dim;

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

s[i].clear();

int a,b;

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

{

scanf("%d",&a);

if(a == 0)

{

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

scanf("%d",&point[i].x[j]);

Insert(i);

}

else

{

scanf("%d",&b);

Delete(b);

}

Query();

}

}

int main()

{

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

solve();

return 0;

}

费马小定理与素数测试

//判定一个数是否是素数的过程：

int power(int a,int e,int m)

{

if(e == 0)

return 1;

if(e == 1)

return a%m;

int t=power(a,e/2,m);

if(e%2 == 1)

return (t\*t\*a)%m;

return (t\*t)%m;

}

bool isprime(int x)

{

const int a[4]={2,3,5,7};

for(int i=0;i<4;i++)

if(power(a[i],x-1,x) != 1)

return false;

return true;

}

注意这是一个概率算法。

其判否的正确性可以保证：即函数返回false，则x一定不是素数；

但是判是的正确性却不是100%：若函数返回true，则x只是很可能是素数。

但是有一个结论：在int范围内，a只需要枚举2、3、5、7就一定能保证完全的正确性。

傅里叶快速变换求两数乘积

struct complex

{

double r,i;

complex(double real=0.0,double image=0.0){

r=real; i=image;

}

// 以下为三种虚数运算的定义

complex operator + (const complex o){

return complex(r+o.r,i+o.i);

}

complex operator - (const complex o){

return complex(r-o.r,i-o.i);

}

complex operator \* (const complex o){

return complex(r\*o.r-i\*o.i,r\*o.i+i\*o.r);

}

}x1[N],x2[N];

char a[N/2],b[N/2];

int sum[N]; // 结果存在sum里

void brc(complex \*y,int l) //二进制平摊反转置换 O(logn)

{

register int i,j,k;

for(i=1,j=l/2;i<l-1;i++)

{

if(i<j) swap(y[i],y[j]); // 交换互为下标反转的元素,i<j保证只交换一次

k=l/2;

while(j>=k) // 由最高位检索，遇1变0，遇0变1，跳出

{

j-=k;

k/=2;

}

if(j<k) j+=k;

}

}

void fft(complex \*y,int l,double on) // FFT O(nlogn),其中on==1时为DFT，on==-1为IDFT

{

register int h,i,j,k;

complex u,t;

brc(y,l); // 调用反转置换

for(h=2;h<=l;h<<=1) // 控制层数

{

// 初始化单位复根

complex wn(cos(on\*2\*PI/h),sin(on\*2\*PI/h));

for(j=0;j<l;j+=h) // 控制起始下标

{

complex w(1,0); // 初始化螺旋因子

for(k=j;k<j+h/2;k++) // 配对

{

u=y[k];

t=w\*y[k+h/2];

y[k]=u+t;

y[k+h/2]=u-t;

w=w\*wn; // 更新螺旋因子

} // 据说上面的操作叫蝴蝶操作…

}

}

if(on==-1)

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

y[i].r/=l; // IDFT

}

void solve()

{

int l1,l2,l;

register int i;

l1=strlen(a);

l2=strlen(b);

l=1;

while(l<l1\*2 || l<l2\*2) l<<=1; // 将次数界变成2^n

// 配合二分与反转置换

for(i=0;i<l1;i++) // 倒置存入

{

x1[i].r=a[l1-i-1]-'0';

x1[i].i=0.0;

}

for(;i<l;i++) x1[i].r=x1[i].i=0.0;

// 将多余次数界初始化为0

for(i=0;i<l2;i++)

{

x2[i].r=b[l2-i-1]-'0';

x2[i].i=0.0;

}

for(;i<l;i++) x2[i].r=x2[i].i=0.0;

fft(x1,l,1); // DFT(a)

fft(x2,l,1); // DFT(b)

for(i=0;i<l;i++) x1[i]=x1[i]\*x2[i]; // 点乘结果存入a

fft(x1,l,-1); // IDFT(a\*b)

for(i=0;i<l;i++) sum[i]=int(x1[i].r+0.5); // 四舍五入

for(i=0;i<l;i++) // 进位

{

sum[i+1]+=sum[i]/10;

sum[i]%=10;

}

l=l1+l2-1;

while(sum[l]<=0 && l>0) l--; // 检索最高位

for(i=l;i>=0;i--) putchar(sum[i]+'0'); // 倒序输出

putchar('\n');

}

高次幂取模

高次幂取模公式：

a^b%c = a^(b%phi(c))%c

高斯消元（浮点数）

#define eps 1e-9

const int MAXN = 220;

double a[MAXN][MAXN], x[MAXN];//a存储方程的左边的矩阵，x存储等式右边的值，求解之后x存的就是结果

int equ, var;//方程数和未知数个数

//返回0表示无解，1表示有解

int Gauss() {

int i, j, k, col, max\_r;

for (k = 0, col = 0; k < equ && col < var; k++, col++) {

max\_r = k;

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

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

max\_r = i;

if (fabs(a[max\_r][col]) < eps)

return 0;

if (k != max\_r) {

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

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

swap(x[k], x[max\_r]);

}

x[k] /= a[k][col];

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

a[k][j] /= a[k][col];

a[k][col] = 1;

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

if (i != k) {

x[i] -= x[k] \* a[i][k];

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

a[i][j] -= a[k][j] \* a[i][col];

a[i][col] = 0;

}

}

return 1;

}

矩阵乘法

int a[N][N],b[N][N],r[N][N];

//一般做法

void Mat\_mul(int n)

{

MEM(r);

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

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

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

r[i][j]=r[i][j]+a[i][k]\*b[k][j];

}

//稍加优化

void Mat\_mul(int n)

{

MEM(r);

for(int k=0;k<n;k++) //k层循环放最外面

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

{

if(!a[i][k]) continue; //加剪枝

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

r[i][j]+=a[i][k]\*b[k][j];

}

}

//带取模的矩阵乘法优化

void mat\_mul(int n,int pri)

{

int i,j,k,L,\*p2;

int tmp[N],con;

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

{

memset(tmp,0,sizeof(tmp));

for(k=0,L=(n&~15);k<L;k++)

{

con=a[i][k];

for(j=0,p2=b[k];j<n;j++,p2++)

tmp[j]+=con\*(\*p2);

if((k&15) == 15)

{

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

tmp[j]%=pri;

}

}

for(;k<n;k++)

{

con=a[i][k];

for(j=0,p2=b[k];j<n;j++,p2++)

tmp[j]+=con\*(\*p2);

}

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

r[i][j]=tmp[j]%pri;

}

}

矩阵快速幂

const int n;

struct Matrix{

LL v[N][N];

};

Matrix Mat\_mul(Matrix m1,Matrix m2) //矩阵相乘

{

Matrix c;

MEM(c.v);

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

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

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

c.v[i][j]=(c.v[i][j]+(m1.v[i][k]\*m2.v[k][j])%MOD)%MOD;

return c;

}

Matrix Mpow(Matrix A,int n) //矩阵快速幂

{

Matrix c,x=A;

MEM(c.v);

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

c.v[i][i]=1;

while(n)

{

if(n & 1)

c=Mat\_mul(c,x);

n>>=1;

x=Mat\_mul(x,x);

}

return c;

}

//定义01矩阵tmp

void init(Matrix &tmp)

{

MEM(tmp.v);

//tmp.v[0][0]=3;tmp.v[0][1]=1;tmp.v[1][0]=1;tmp.v[1][1]=0;

}

void work(int n)

{

Matrix tmp,ans;

init(tmp);

ans=Mpow(tmp,n);

}

卡特兰数

#define N 1000

#define MOD 1000000007

LL f[N+10];

LL mod\_exp(LL a,LL b)

{

LL ans=1;

while(b)

{

if(b & 1)

ans=ans\*a%MOD;

a=a\*a%MOD;

b/=2;

}

return ans;

}

void init()

{

f[0]=1;f[1]=1;

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

{

LL tt=mod\_exp(i+1,MOD-2);

LL t=((4\*i-2)\*f[i-1])%MOD;

f[i]=(t\*tt)%MOD;

}

}

//1,2,5,14,42,132,429,1430,4862,16796,58786,208012

快速乘取余a\*b%c

//a\*b%c

LL mod\_mul(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(蒙哥马利算法)

LL Mod\_exp(LL a,LL b,LL c)

{

LL ans=1;

while(b)

{

if(b & 1)

ans=ans\*a%c;

a=(a%c)\*(a%c)%c;

b/=2;

}

return ans;

}

扩展欧几里德

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

//如果求x的最小正整数解，如果x不大于0，则令x=x+b/g, y=y-a/g

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

{

if(b == 0)

{

x=1;y=0;

return a;

}

int t,d;

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

t=x;x=y;y=t-a/b\*y;

return d;

}

//线性方程定理，利用一组解求出a\*x+b\*y=gcd(a,b)的其它解，k可为任意整数

void solve(int a,int b)

{

int x,y,g,x1,y1;

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

for(int k=0;k<10;k++)

{

x1=x+b/g\*k;

y1=y-a/g\*k;

printf("%d %d\n",x1,y1);

}

}

拉宾-米勒测试

#define LL \_\_int64

//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);

}

//拉宾-米勒测试，是素数返回1，否则返回0

int rabin\_miller(int n)

{

if (n==2)

return 1;

if ((n%2)==0)

return 0;

int j,s=50;

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;

}

牛顿迭代法求开方

//原理是二分

double NT\_sqrt(double n)

{

double m = 1;

while(fabs(m-(m+n/m)/2)>1e-6)

m=(m+n/m)/2;

return m;

}

欧拉函数

欧拉函数定义：

对正整数n，欧拉函数是少于或等于n的数中与n互质的数的数目。

#define N 1000000

LL prime[N];//存素数

bool f[N];

LL Ou[N+2];//存欧拉函数

void fun()

{

// //打素数表和欧拉函数表，前N个

\_\_int64 i,j,pNum=0;

memset(f,false,sizeof(f));

Ou[1]=1;

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

{

if(!f[i])

{

prime[pNum++]=i;

Ou[i]=i-1;

}

for(j=0;j<pNum && prime[j]\*i<=N;j++)

{

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

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

{

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

break;

}

else

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

}

}

}

void fun()

{

//下面求的素数表和欧拉函数表，是根据欧拉函数定义来求解，上面部分是经过改进后的求法（效率较高）

LL i,j,pNum=0;

memset(f,true,sizeof(f));

f[0]=f[1]=false;

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

if(f[i])

for(j=i\*i;j<=N;j+=i)

f[j]=false;

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

Ou[i]=i;

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

if(f[i])

{

prime[pNum++]=i;//如果不需要素数表，可以把这步省去

for(j=i;j<=N;j+=i)//根据欧拉函数的定义来更新

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

}

}

//递推求欧拉函数值

#define N 1000000

int phi[N+10];

void init()

{

int i,j;

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

phi[i]=i;

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

phi[i]/=2;

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

if(phi[i]==i)

{

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

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

}

}

//求单个值的欧拉函数值

int Getphi(int n)

{

int i,phi;

for(i=2,phi=n;i\*i<=n;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;

}

判断n(最大10^18)是否能被一个数的平方整除

问n<=10^18是否能被一个数的平方整除。

将n分解为：p1p2pk 其中pi为素数，且pi<=pi+1（这里可以等于）若n能被一个数的平方整除，它肯定能被一个素因子的平方p^2整除，我们找到最小的这个p即可

1.如果p不是最后的素因子，在p之后的素数pi>=p，这时就有n>=p^3了，即p<=n^(1/3)，所以枚举n^(1/3)内的素数看是否p^2|n即可，而n^(1/3)<=10^6

2.如果p是最后的素因子，所以n的形式就是p1p2p'p^2，而p'<=p，所以P'^3<=n，p'<=n^(1/3) 所以可以先将n的这些素因子p1,p2,,p'给去掉，p'<=n^(1/3)，所以用n^(1/3)内的素数去除即可最后只剩下n'=p^2，通过判断一个数是否为完全平方数即可

算法就是预处理10^6内的素数，用这些素数去除n，发现有p^2能整除的return即可

否则判断剩下的这个n'是否为完全平方数

int tot,prime[N+10];

int f[N+10];

void Getprime()

{

memset(f,0,sizeof(f));

tot=0;

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

{

if(f[i] == 0)

prime[tot++]=f[i]=i;

for(int j=0;j<tot && prime[j]<=f[i] && i\*prime[j]<=N;j++)

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

}

}

int jud(LL n)

{

LL a=(LL)sqrt(n\*1.0);

if(a\*a == n) return 1;

return 0;

}

int fun(LL n)

{

for(int i=0;i<tot && prime[i]\*prime[i]\*prime[i]<=n;i++)

{

int c=0;

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

{

n/=prime[i];

c++;

if(c >= 2) return 1;

}

}

if(jud(n)) return 1;

return 0;

}

int main()

{

LL n;

Getprime();

while(sf64(n)!=EOF)

{

if(fun(n)) pfs("Yes");

else pfs("No");

}

return 0;

}

求1到n的n个数的最小公倍数

求lcm(1,2,3,...,n)

1. 先把所有小于n的素数乘在一起，得到ans。（这一步也可以先打表再二分）

2. 考虑素数a的k次方的数，找到小于n的最大a的k次方数，ans乘上k-1个a。

例如n=9,a=2,2的k次方数最大为8=2^3，k=3，需要在ans上乘上2个2.

int n;

int tot,prime[N+10],f[N+10];

void Getprime()

{

memset(f,0,sizeof(f));

tot=0;

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

{

if(f[i] == 0)

prime[tot++]=f[i]=i;

for(int j=0;j<tot && prime[j]<=f[i] && i\*prime[j]<=N;j++)

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

}

}

void fun(){

LL ans=1;

for(int i=0;i<tot && prime[i] <= n;i++)

ans=ans\*prime[i]%MOD;

for(int i=0;i<tot && prime[i]\*prime[i]<=n;i++){

LL y=prime[i]\*prime[i];

while(y <= n) {

ans=ans\*prime[i]%MOD;

y\*=prime[i];

}

}

pf64(ans);

}

void solve(){

sf(n);

fun();

}

求解x^2=a mod (n)

//求解x^2 ≡ a mod (n), n为素数.

//有解时.返回较小解x, 另外一解为 n-x.

LL Mod\_exp(LL a,LL b,LL c)

{

LL ans=1;

while(b)

{

if(b & 1)

ans=ans\*a%c;

a=(a%c)\*(a%c)%c;

b/=2;

}

return ans;

}

LL ModSqrt(LL a,LL n)

{

int b,k,i,x;

if(n == 2)

return a%n;

if(Mod\_exp(a,(n-1)/2,n) == 1)

{

if(n%4==3)

x=Mod\_exp(a,(n+1)/4,n);

else

{

for(b=1;Mod\_exp(b,(n-1)/2,n)==1;b++);

i=(n-1)/2;k=0;

do{

i/=2;k/=2;

if((Mod\_exp(a,i,n)\*(LL)Mod\_exp(b,k,n)+1)%n == 0)

k+=(n-1)/2;

}while(i%2 == 0);

x=(Mod\_exp(a,(i+1)/2,n)\*(LL)Mod\_exp(b,k/2,n))%n;

}

if(x\*2>n)

x=n-x;

return x;

}

return -1;

}

求佩尔方程的第k解

定义：形如x^2-n\*y^2=1的方程

方程性质：

如果n是完全平方数，则方程无解；否则方程有无穷多组正整数解

每组解(xi,yi)存在这样的递推关系式：

xi = xi-1 \* x1 + n \* yi-1 \* y1;

yi = xi-1 \* y1 + yi-1 \* x1.

要求方程第k个解，可以先求出最小正整数解(p,q)，然后利用上面的递推式，构造2\*2的矩阵tmp(构造如下)，用矩阵快速幂解决。

矩阵tmp构造如下：

tmp.v[0][0]=p;tmp.v[0][1]=q;tmp.v[1][0]=n\*q;tmp.v[1][1]=p;

第k个解=[p,q]\*(tmp^(k-1));

求最小正整数解(p,q)模板：

struct Pell{

LL p,q;

};

struct Node{

LL g,h;

};

Pell Getminpell(LL n)

{

Pell s[4];

Node w[4];

int a[4];

s[0].p=0; s[0].q=1;

s[1].p=1; s[1].q=0;

a[0]=(LL)floor(sqrt(n\*1.0));

a[2]=a[0];

w[1].g=0;w[1].h=1;

while(1)

{

w[2].g = -w[1].g+a[2]\*w[1].h;

w[2].h = (n-w[2].g\*w[2].g)/w[1].h;

a[3] = (LL)floor( (double)(w[2].g+a[0])/w[2].h );

s[2].p = a[2]\*s[1].p+s[0].p;

s[2].q = a[2]\*s[1].q+s[0].q;

if( (s[2].p\*s[2].p-n\*s[2].q\*s[2].q) == 1 && s[2].p>0 && s[2].q>0 )

return s[2];

w[0]=w[1];w[1]=w[2];

a[2]=a[3];

s[0]=s[1];s[1]=s[2];

}

}

void solve(LL n)

{

LL t=(LL)sqrt(n\*1.0);

if(t\*t == n)

{

printf("No answer.\n");

return ;

}

Pell ans=Getminpell(n);

printf("%I64d %I64d\n",ans.p,ans.q);

}

日期相关函数

//日期函数

char str\_mon[15][15]={"January","February","March","April","May","June","July","August","September","October","November","December"};

char str\_week[15][15]={"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"};

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

struct date{

int year,month,day;

};

//判闰年

inline int leap(int year)

{

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

}

//判合法性

inline int legal(date a)

{

if (a.month<0 || a.month>12)

return 0;

if (a.month==2)

return a.day>0 && a.day<=28+leap(a.year);

return a.day>0 && a.day<=days[a.month-1];

}

//比较日期大小

inline int datecmp(date a,date b)

{

if (a.year != b.year)

return a.year-b.year;

if (a.month != b.month)

return a.month-b.month;

return a.day-b.day;

}

//返回指定日期是星期几

int weekday(date a)

{

int tm=a.month>=3?(a.month-2):(a.month+10);

int ty=a.month>=3?a.year:(a.year-1);

return (ty+ty/4-ty/100+ty/400+(int)(2.6\*tm-0.2)+a.day)%7;

}

//日期转天数偏移

int date2int(date a)

{

int ret=a.year\*365+(a.year-1)/4-(a.year-1)/100+(a.year-1)/400,i;

days[1]+=leap(a.year);

for (i=0;i<a.month-1;ret+=days[i++]);

days[1]=28;

return ret+a.day;

}

//天数偏移转日期

date int2date(int a)

{

date ret;

ret.year=a/146097\*400;

for(a%=146097;a>=365+leap(ret.year);a-=365+leap(ret.year),ret.year++);

days[1]+=leap(ret.year);

for(ret.month=1;a>=days[ret.month-1];a-=days[ret.month-1],ret.month++);

days[1]=28;

ret.day=a+1;

return ret;

}

数学公式

1^4+2^4+3^4+4^4+……+n^4 = n(n+1)(2n+1)(3n^2+3n-1)/30

1+1/2+1/3+……+1/n=?

如果n小，则直接循环求解（或打表）

如果n大，则结果近似等于ln(n)+C,C是欧拉常数

C=0.57721566490153286060651209

素数判定

int jud(int n)

{

if(n<2) return 0;

for(int k=2;k\*k<=n;k++)

if(n%k==0)

return 0;

return 1;

}

素数筛选法

(Eratosthenes埃拉托斯特尼筛选法)

int tot,prime[N+10],f[N+10];

//高效方法，时间复杂度O(N)

void Getprime()

{

memset(f,0,sizeof(f));

tot=0;

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

{

if(f[i] == 0)

prime[tot++]=f[i]=i;

for(int j=0;j<tot && prime[j]<=f[i] && i\*prime[j]<=N;j++)

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

}

}

//朴素方法，时间复杂度O(NlogN)

void Getprime()

{

memset(f,0,sizeof(f));

tot=0;

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

{

if(f[i] == 0)

{

prime[tot++]=i;

for(int j=1;j\*i<=N;j++)

f[i\*j]=1;

}

}

}

线性同余式方程

//线性同余式定理解方程 a\*x = c(mod m)

void mod\_equ(int a,int c,int m)

{

int g,x,y,x1,ans;

g=extend\_gcd(a,m,x,y);

if(c%g!=0)

return ;

x1=((c/g\*x)%m+m)%m;

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

{

ans=x1+m/g\*i;

ans=(ans+m)%m;

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

}

}

质因数分解

Pro 1：分解质因数，将数n所有因子（包括重复的）存在p数组里，个数为plen

void Getfac(int n)

{

plen=0;

for(int i=0;prime[i]\*prime[i] <= n ;i++)

{

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

{

p[plen++]=prime[i];

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

n/=prime[i];

}

if(n == 1)

break;

}

if(n > 1)

p[plen++]=n;

}

Pro 2：快速获取n的所有因子（n很大）

int len\_fac,len\_tot;

LL resfac[N],fac[N],pfac[N];

//分解质因数，将数n的质因子（不包括重复的）记录在fac数组里，因子的幂记录在pfac数组里，个数为len\_fac

void Getfac(int n)

{

int s;

len\_fac=0;

for(int i=0;i<tot && prime[i]\*prime[i] <= n;i++)

{

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

{

s=0;

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

{

n/=prime[i];

s++;

}

fac[len\_fac]=prime[i];pfac[len\_fac++]=s;

}

}

if(n > 1)

{

fac[len\_fac]=n;

pfac[len\_fac++]=1;

}

}

//dfs构造数n的所有因子并记录在resfac数组里，个数为len\_tot

void dfs(int tmp,int num)

{

if(tmp == len\_fac)

{

resfac[len\_tot++]=num;

return ;

}

int tt=1;

for(int i=0;i<=pfac[tmp];i++)

{

dfs(tmp+1,num\*tt);

tt=tt\*fac[tmp];

}

}

void solve(int n)

{

Getprime();//素数筛选打印素数表

Getfac(n);

len\_tot=0;

dfs(0,1);

}

质因数分解求n个数的最小公倍数

题面：

给n个数ai，求n个数的最小公倍数s，求s%M的结果。ai,n<=10^6，M=3\*2^30+1.

#define M 3221225473ll

int n,f[N];

int p[N];

int tot,prime[NN+10],ff[NN+10];

void Getprime()

{

memset(ff,0,sizeof(ff));

tot=0;

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

{

if(ff[i] == 0)

prime[tot++]=ff[i]=i;

for(int j=0;j<tot && prime[j]<=ff[i] && i\*prime[j]<=NN;j++)

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

}

}

LL mod\_exp(LL a,LL b,LL c)

{

LL ans=1;

while(b)

{

if(b & 1)

ans=ans\*a%c;

a=a\*a%c;

b/=2;

}

return ans;

}

void fun(){

MEM(p);

for(int t=1;t<=n;t++){

for(int i=0;i<tot && prime[i]\*prime[i]<=f[t];i++){

int c=0;

while(f[t]%prime[i] == 0){

c++;

f[t]/=prime[i];

}

p[prime[i]]=max(p[prime[i]],c);

}

if(f[t] > 1){

p[f[t]]=max(1,p[f[t]]);

}

}

LL ans=1;

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

if(p[i]){

ans=ans\*mod\_exp(i,p[i],M)%M;

}

}

pf64(ans);

}

int main(){

Getprime();

int T;

sf(T);

while(T--){

sf(n);

fr(n) sf(f[i]);

fun();

}

return 0;

}

质因数分解求大数阶乘对p取余

//还可以求类似这样的问题：(n+1)!/(m!)

int p[M];

bool f[N];

int n,tot,len,prime[M];

//打印素数表

void fun()

{

memset(f,0,sizeof(f));

tot=0;

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

{

if(f[i] == 0)

prime[tot++]=f[i]=i;

for(int j=0;j<tot && prime[j]<=f[i] && i\*prime[j]<=N;j++)

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

}

}

//对某个数的阶乘分解质因数，temp为0表示数的阶乘在分子上的，为1表示在分母上

void fun2(int t,int temp)

{

int tmp,i;

for(i=0;i<tot && prime[i]<=t;i++)

{

tmp=t;

while(tmp)

{

if(temp == 0)

p[i]+=tmp/prime[i];

else

p[i]-=tmp/prime[i];

tmp/=prime[i];

}

}

len=max(len,i);

}

//快速幂取余

LL mod\_exp(LL a,LL b,LL c)

{

LL ans=1;

while(b)

{

if(b & 1)

ans=ans\*a%c;

a=a\*a%c;

b/=2;

}

return ans;

}

LL Get()

{

LL tmp,ans=1;

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

{

tmp=mod\_exp((LL)prime[i],(LL)p[i],MOD);

ans=ans\*tmp%MOD;

}

return ans;

}

void solve()

{

len=-1;MEM(p);

fun();

fun2(n,0);

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

}

中国剩余定理

//b[]任意两个数不互质的情况，用线性同余式定理求解

int num[15],rem[15];

int Extend\_gcd(int a,int b,int &x,int &y) //扩展欧几里得算法

{

int d;

if(b==0)

{

x=1;y=0;

return a;

}

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

y-=a/b\*x;

return d;

}

//返回0的时候如果有需要特判下，p记录所有除数的最小公倍数

int Chinese\_Remainder(int n,int num[],int remain[],int &p)

{

int a,b,c,d,x,y,remain1,remain2;

a=num[0];remain1=remain[0];

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

{

b=num[i];remain2=remain[i];c=remain2-remain1;

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

if(c%d)

return -1; //不整除表示无解

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

remain1=remain1+a\*x;

a=(a\*b)/d;

remain1=(remain1%a+a)%a;

}

p=a;

return remain1;

}

最大公约数和最小公倍数

int gcd(int a,int b){return !b?a:gcd(b,a%b);}

int lcm(int a,int b){return a/gcd(a,b)\*b;}

### 五、博弈

Crazy Nim

变形的Nim，有三堆石子，各有a,b,c个石子数，每次任意选一堆取任意数量的石子，要求取完后不能使任意两堆石子数相同，按最优策略进行，问先手是否必胜。

结论：t=(a+1)^(b+1)^(c+1)，如果t大于0，先手必胜，否则必败。

Every-SG游戏

参考贾志豪的《组合游戏略述——浅谈SG游戏的若干拓展及变形》。

Every-SG 游戏定义：

还没有结束的单一游戏，游戏者必须做出决策，规则与普通SG游戏相同。即N组SG游戏同时进行。

游戏策略：

对于单一游戏，如果能赢，则希望玩的时间越久越好；如果为输，则希望玩的时间越短越好。

所以，我们需要知道的是：单一游戏如果为赢，最长能玩多久到终止状态；如果为输，最短能玩多久到终止状态。

我们用step(v)来表示这个值，则有：

step(v)=0, v为终止状态

step(v)=max(step(u))+1, sg(v)>0 && sg(u)=0

step(v)=min(step(u))+1, sg(v)=0, u为v的后继状态.

【Every-SG定理】

先手必胜当且仅当单一游戏中最大的step为奇数。

//hdu 3595

有N个游戏同时进行，每个游戏有两堆石子，每次从个数多的堆中取走数量小的数量的整数倍的石子。不能取为输，N个游戏同时进行，每个游戏必须操作，除非游戏结束。

int f[1010][1010];

int step[1010][1010];

int sg(int a,int b)

{

if(a > b) swap(a,b);

if(f[a][b] != -1)

return f[a][b];

if(a==0 || b==0)

return step[a][b]=step[b][a]=f[a][b]=f[b][a]=0;

int mn=INF,mx=-INF;

for(int k=1;k\*a <= b;k++)

{

if(sg(a,b-k\*a) == 0)

mx=max(mx,step[a][b-k\*a]);

else

mn=min(mn,step[a][b-k\*a]);

}

if(mx != -INF)

{

step[a][b]=step[b][a]=mx+1;

return f[a][b]=f[b][a]=1;

}

step[a][b]=step[b][a]=mn+1;

return f[a][b]=f[b][a]=0;

}

int main()

{

int n,a,b;

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

while(sf(n)!=EOF)

{

int ans=0;

while(n--)

{

sf2(a,b);

sg(a,b);

ans=max(ans,step[a][b]);

}

if(ans&1) pfs("MM");

else pfs("GG");

}

return 0;

}

SG函数

定义：对于一个递增有界的图G(X, F)来说，SG函数式定义在X上的函数，函数值是非负整数。

g(x)的值等于x的所有后继SG函数值中没有出现过的最小非负整数。

对于递增有界的图，SG函数是唯一的、有界的。

所有的终止状态x，g(x)=0.

SG函数的几条性质：

如果x是终止状态，那么g(x)=0;

一个状态x，如果g(x)≠0，那么一定存在一个x的后继y，使得g(y)=0;

一个状态x，如果g(x)=0,那么所有x的后继y，都有g(y)≠0.

//hdu1536&1944

int s[110],f[10010];

int sg(int a,int k)

{

int mex[110];

MEM(mex);

if(f[a] != -1)

return f[a];

if(a < s[0])

return f[a]=0;

for(int i=0;i<k && a>=s[i];i++)

{

int tmp=sg(a-s[i],k);

mex[tmp]=1;

}

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

if(!mex[i])

return f[a]=i;

}

void solve(int k)

{

int n,ans,h;

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

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

sort(s,s+k);

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

scanf("%d",&n);

ans=0;

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

{

scanf("%d",&h);

ans=ans^sg(h,k);

}

if(!ans)

printf("No\n");

else

printf("Yes");

}

博弈搜索树

博弈搜索树：

如果后继状态中有必败态，则当期状态必胜；

如果后继状态中没有必败态，则当期状态必败。

//hdu 4155

有1到6各4张的牌堆，两个轮流取，取出来的牌累加，谁取出的牌后累加大于等于31即为输。

现已知游戏进行的前几个回合，问继续游戏，谁能获胜。

char str[32];

int p[12];

int dfs(int tmp)

{

if(tmp >= 31)

return 0;

for(int i=1;i<=6;i++)

{

if(p[i] && tmp+i <= 31)

{

p[i]--;

int t=dfs(tmp+i);

p[i]++;

if(t == 0)

return 1;

}

}

return 0;

}

int main()

{

while(sfs(str)!=EOF)

{

printf("%s",str);

pfk;

for(int i=1;i<7;i++)

p[i]=4;

int len=LEN(str);

int t=0;

fr(len){

p[str[i]-'0']--;

t=t+str[i]-'0';

}

int r=dfs(t);

if(len&1)

{

if(r == 1) pfs("B");

else pfs("A");

}

else

{

if(r == 1) pfs("A");

else pfs("B");

}

}

return 0;

}

/\*

Input:

356656

35665

3566

111126666

552525

Output:

356656 B

35665 B

3566 A

111126666 A

552525 A

\*/

取火柴游戏

取火柴的游戏

题目1：今有若干堆火柴，两人依次从中拿取，规定每次只能从一堆中取若干根，可将一堆全取走，但不可不取，最后取完者为胜，求必胜的方法。

题目2：今有若干堆火柴，两人依次从中拿取，规定每次只能从一堆中取若干根，

可将一堆全取走，但不可不取，最后取完者为负，求必胜的方法。

先解决第一个问题吧。

定义：若所有火柴数异或为0，则该状态被称为利他态，用字母T表示；否则为利己态，用S表示。

[定理1]：对于任何一个S态，总能从一堆火柴中取出若干个使之成为T态。

证明：

若有n堆火柴，每堆火柴有A(i)根火柴数，那么既然现在处于S态，c = A(1) xor A(2) xor … xor A(n) > 0;把c表示成二进制，记它的二进制数的最高位为第p位，则必然存在一个A(t),它二进制的第p位也是1。（否则，若所有的A(i)的第p位都是0，这与c的第p位就也为0矛盾）。

那么我们把x = A(t) xor c,则得到x < A(t).这是因为既然A(t)的第p位与c的第p位同为1,那么x的第p位变为0,而高于p的位并没有改变。所以x < A(t).而

A(1) xor A(2) xor … xor x xor … xor A(n)

= A(1) xor A(2) xor … xor A(t) xor c xor … xor A(n)

= A(1) xor A(2) xor… xor A(n) xor A(1) xor A(2) xor … xor A(n)

= 0

这就是说从A(t)堆中取出 A(t) – x 根火柴后状态就会从S态变为T态。证毕

[定理2]：T态，取任何一堆的若干根，都将成为S态。

证明：用反证法试试。

若

c = A(1) xor A(2) xor … xor A(i) xor … xor A(n) = 0；

c’ = A(1) xor A(2) xor … xor A(i’) xor c xor … xor A(n) = 0;

则有

c xor c’ = A(1) xor A(2) xor … xor A(i) xor … xor A(n) xor A(1) xor A(2) xor … xor A(i’) xor c xor … xor A(n) = A(i) xor A(i’) =0

进而推出A(i) = A(i’)，这与已知矛盾。所以命题得证。

[定理 3]：S态，只要方法正确，必赢。

最终胜利即由S态转变为T态，任何一个S态，只要把它变为T态，（由定理1，可以把它变成T态。）对方只能把T态转变为S态(定理2)。这样，所有S态向T态的转变都可以有己方控制，对方只能被动地实现由T态转变为S态。故S态必赢。

[定理4]：T态，只要对方法正确，必败。

由定理3易得。

接着来解决第二个问题。

定义：若一堆中仅有1根火柴，则被称为孤单堆。若大于1根，则称为充裕堆。

定义：T态中，若充裕堆的堆数大于等于2，则称为完全利他态，用T2表示；若充裕堆的堆数等于0，则称为部分利他态，用T0表示。

孤单堆的根数异或只会影响二进制的最后一位，但充裕堆会影响高位（非最后一位）。一个充裕堆，高位必有一位不为0，则所有根数异或不为0。故不会是T态。

[定理5]：S0态，即仅有奇数个孤单堆，必败。T0态必胜。

证明：

S0态，其实就是每次只能取一根。每次第奇数根都由己取，第偶数根都由对方取，所以最后一根必己取，败。同理, T0态必胜

[定理6]：S1态，只要方法正确，必胜。

证明：

若此时孤单堆堆数为奇数，把充裕堆取完；否则，取成一根。这样，就变成奇数个孤单堆，由对方取。由定理5，对方必输。己必胜。

[定理7]：S2态不可转一次变为T0态。

证明：

充裕堆数不可能一次由2变为0。得证。

[定理8]：S2态可一次转变为T2态。

证明：

由定理1，S态可转变为T态，态可一次转变为T态，又由定理6，S2态不可转一次变为T0态，所以转变的T态为T2态。

[定理9]：T2态，只能转变为S2态或S1态。

证明：

由定理2，T态必然变为S态。由于充裕堆数不可能一次由2变为0，所以此时的S态不可能为S0态。命题得证。

[定理10]：S2态，只要方法正确，必胜.

证明：

方法如下：

1）S2态，就把它变为T2态。（由定理8）

2）对方只能T2转变成S2态或S1态（定理9）

若转变为S2, 转向1）

若转变为S1, 这己必胜。（定理5）

[定理11]：T2态必输。

证明：同10。

综上所述，必输态有： T2,S0

必胜态： S2,S1,T0.

两题比较：

第一题的全过程其实如下：

S2->T2->S2->T2-> …… ->T2->S1->T0->S0->T0->……->S0->T0(全0)

第二题的全过程其实如下：

S2->T2->S2->T2-> …… ->T2->S1->S0->T0->S0->……->S0->T0(全0)

下划线表示胜利一方的取法。 是否发现了他们的惊人相似之处。

我们不难发现(见加黑部分)，S1态可以转变为S0态（第二题做法），也可以转变为

T0（第一题做法）。哪一方控制了S1态，他即可以有办法使自己得到最后一根（转变为

T0）,也可以使对方得到最后一根（转变为S0）。

所以，抢夺S1是制胜的关键！

为此，始终把T2态让给对方，将使对方处于被动状态，他早晚将把状态变为S1.

第二题也可以用SJ定理解决

【SJ定理】

对于任意一个Anti-SG游戏，如果我们规定当局面中所有的单一游戏的SG值为0时，游戏结束，则先手必胜当且仅当：

（1）游戏的SG函数不为0且游戏中某个单一游戏的SG函数大于1；

（2）游戏的SG函数为0且游戏中没有单一游戏的SG函数大于1。

void solve(int n)

{

int a,s=0,num=0;

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

{

scanf("%d",&a);

s=s^a;

if(a > 1)

num++;

}

if(s == 0)

{

if(num == 0)

printf("Yes\n");

else

printf("No\n");

}

else

{

if(num == 0)

printf("No\n");

else

printf("Yes\n");

}

}

三类经典博弈

（一）巴什博奕（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。

显然，如果n=m+1，那么由于一次最多只能取m个，所以，无论先取者拿走多少个，后取者都能够一次拿走剩余的物品，后者取胜。因此我们发现了如何取胜的法则：如果n=（m+1）r+s，（r为任意自然数，s≤m),那么先取者要拿走s个物品，如果后取者拿走k（≤m)个，那么先取者再拿走m+1-k个，结果剩下（m+1）（r-1）个，以后保持这样的取法，那么先取者肯定获胜。总之，要保持给对手留下（m+1）的倍数，就能最后获胜。

（二）威佐夫博奕（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。

这种情况下是颇为复杂的。我们用（ak，bk）（ak ≤ bk ,k=0，1，2，…,n)表示

两堆物品的数量并称其为局势，如果甲面对（0，0），那么甲已经输了，这种局势我们

称为奇异局势。前几个奇异局势是：（0，0）、（1，2）、（3，5）、（4，7）、（6，

10）、（8，13）、（9，15）、（11，18）、（12，20）。

可以看出,a0=b0=0,ak是未在前面出现过的最小自然数,而 bk= ak + k，奇异局势有

如下三条性质：

1。任何自然数都包含在一个且仅有一个奇异局势中。

由于ak是未在前面出现过的最小自然数，所以有ak > ak-1 ，而 bk= ak + k > ak

-1 + k-1 = bk-1 > ak-1 。所以性质1。成立。

2。任意操作都可将奇异局势变为非奇异局势。

事实上，若只改变奇异局势（ak，bk）的某一个分量，那么另一个分量不可能在其

他奇异局势中，所以必然是非奇异局势。如果使（ak，bk）的两个分量同时减少，则由

于其差不变，且不可能是其他奇异局势的差，因此也是非奇异局势。

3。采用适当的方法，可以将非奇异局势变为奇异局势。

假设面对的局势是（a,b），若 b = a，则同时从两堆中取走 a 个物体，就变为了

奇异局势（0，0）；如果a = ak ，b > bk，那么，取走b – bk个物体，即变为奇异局

势；如果 a = ak ， b < bk ,则同时从两堆中拿走 ak – ab – ak个物体,变为奇异局

势（ ab – ak , ab – ak+ b – ak）；如果a > ak ，b= ak + k,则从第一堆中拿走多余

的数量a – ak 即可；如果a < ak ，b= ak + k,分两种情况，第一种，a=aj （j < k）

,从第二堆里面拿走 b – bj 即可；第二种，a=bj （j < k）,从第二堆里面拿走 b – a

j 即可。

从如上性质可知，两个人如果都采用正确操作，那么面对非奇异局势，先拿者必胜

；反之，则后拿者取胜。

那么任给一个局势（a，b），怎样判断呢？我们有如下公式：

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…,因此,由ak，bk组成的矩形近

似为黄金矩形，由于2/（1+√5）=（√5-1）/2，可以先求出j=[a（√5-1）/2]，若a=[

j（1+√5）/2]，那么a = aj，bj = aj + j，若不等于，那么a = aj+1，bj+1 = aj+1

+ j + 1，若都不是，那么就不是奇异局势。然后再按照上述法则进行，一定会遇到奇异

局势。

（三）尼姆博奕（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。

这种情况最有意思，它与二进制有密切关系，我们用（a，b，c）表示某种局势，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）。仔细分析一下，（1，2，3）也是奇异局势，无论对手如何拿，接下来都可以变为（0，n，n）的情形。

计算机算法里面有一种叫做按位模2加，也叫做异或的运算，我们用符号（+）表示

这种运算。这种运算和一般加法不同的一点是1+1=0。

任何奇异局势（a，b，c）都有a（+）b（+）c =0。

如果我们面对的是一个非奇异局势（a，b，c），要如何变为奇异局势呢？假设 a < b

< c,我们只要将 c 变为 a（+）b,即可,因为有如下的运算结果: a（+）b（+）(a（+）

b)=(a（+）a)（+）(b（+）b)=0（+）0=0。要将c 变为a（+）b，只要从 c中减去 c-（

a（+）b）即可。

### 六、字符串

KMP

/\*

\* 输入主串和模式串，如果模式串在主串中匹配，则返回模式串在主串中的首位置，否则输出No

\*/

#define N 10001

char s1[N],s2[N];//s1为主串，s2为模式串

int next[N];

void getnext(int next[],char s[])

{

int len,i=0,j=-1;

next[0]=-1;

len=(int)strlen(s);

while(i<len)

{

if(j==-1 || s[i]==s[j])

{

i++;j++;

if(s[i] != s[j])//优化

next[i]=j;

else

next[i]=next[j];

}

else

j=next[j];

}

}

int KMP(char s1[],char s2[],int next[])

{

//返回模式串在主串中出现的首位置

int i,j,len1,len2;

i=0;j=0;

len1=(int)strlen(s1);

len2=(int)strlen(s2);

while(i<len1 && j<len2)

{

if(j<0||s1[i]==s2[j])

{

i++;j++;

}

else

j=next[j];

}

if(j==len2)

return i-j;

return 0;

}

int KMP(char s1[],char s2[],int next[])

{

//返回最长相同的主串后缀和模式串前缀

int i,j,len1,len2;

i=0;j=0;

len1=(int)strlen(s1);

len2=(int)strlen(s2);

while(i<len1)

{

if(j<0 || s1[i]==s2[j])

{

i++;j++;

}

else

j=next[j];

}

return j;

}

int main()

{

int ans;

while(scanf("%s %s",s1,s2)!=EOF)

{

getnext(next,s2);

ans=KMP(s1,s2,next);

if(ans!=0)

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

else

printf("No!\n");

}

return 0;

}

AC自动机

//hdu 2222

const int N=55;

const int M=1000010;

#define BASE 'a'

char key[N],str[M];

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 num; //是否为该单词的最后一个节点

node()

{ //构造函数初始化

fail = NULL;

num = 0;

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

next[i]=NULL;

}

}\*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] - BASE;

if (p->next[index] == NULL)

p->next[index] = new node();

p = p->next[index];

i++;

}

p->num++;

}

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 < kind; 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] - BASE;

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->num != -1)

{

cnt += temp->num;

temp->num = -1;//若每个单词可以匹配多次则去掉

temp = temp->fail;

}

i++;

}

return cnt;

}

int main()

{

int T,n;

scanf("%d",&T);

while(T--)

{

scanf("%d",&n);

//node \*root;

root=new node();

while(n--)

{

scanf("%s",key);

insert(key,root);

}

scanf("%s",str);

build\_ac\_automation(root);

printf("%d\n",query(root,str));

}

return 0;

}

Manacher算法求字符串

最长回文子串

时间复杂度为O(n)

#define N 110010

char s[N\*2],str[N\*2];

int n,p[N\*2];

//将字符串如“abab”处理成新的字符串“$#a#b#a#b#”

void fun()

{

int len=(int)strlen(s);

str[0]='$';str[1]='#';

n=2;

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

{

str[n++]=s[i];

str[n++]='#';

}

str[n]='\0';

}

void Manacher()

{

int i,id,mx=0;

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

{

if(mx > i)

p[i]=min(p[2\*id-i],p[id]+id-i);

else

p[i]=1;

for(;str[i+p[i]] == str[i-p[i]];p[i]++)

;

if(p[i]+i > mx)

{

mx=p[i]+i;

id=i;

}

}

}

void work()

{

int ans=-1;

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

ans=max(ans,p[i]);

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

}

int main()

{

while(scanf("%s",s)!=EOF)

{

fun();

Manacher();

work();

}

return 0;

}

Trie树求字符串所有不相同子串

#define N 1000010

char s[N],temp[N];

int nodeNum;

const int childNum=26,base='a';

struct TrieNode{

struct TrieNode \*child[childNum];

};

void init(TrieNode \*p)

{

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

p->child[i]=NULL;

}

void destroy(TrieNode \*p)

{

if(p!=NULL)

{

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

{

if(p->child[i]!=NULL)

destroy(p->child[i]);

}

}

delete p;

}

void Insert(TrieNode \*root,char \*s)

{

TrieNode \*p=root;

int len=(int)strlen(s);

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

{

if(p->child[s[i]-base]==NULL)

{

p->child[s[i]-base]=new TrieNode();

init(p->child[s[i]-base]);

nodeNum++;

}

p=p->child[s[i]-base];

}

}

int main()

{

int len;

TrieNode \*root;

while(scanf("%s",s)!=EOF)

{

len=(int)strlen(s);

root=new TrieNode;

init(root);

nodeNum=0;

while(len--)

{

strcpy(temp,&s[len]);

Insert(root,temp);

}

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

destroy(root);

}

return 0;

}

后缀数组

后缀数组是将所有后缀子串按从小到大顺序排列。

关于后缀数组的三个数组：sa[i]，ranks[i]，height[i]。

sa[i]表示排第i的后缀首字母在原字符串中位置sa[i]；

rank[i]表示首字母在原字符串中位置为i的后缀排第rank[i]（与sa[i]相对），第一的rank[]值为1，不为0；

height[i]表示排第i的后缀与前一个后缀的最长公共前缀长度，height[]从1开始有值，[1,n)。

求后缀数组常见有两种算法，倍增算法(O(nlogn)); DC3算法(O(n)).

char s[N];

int n, sa[4\*N], rank[N], height[N];

int buf[4\*N], ct[N], sx[N], sax[N];

inline bool leq(int a, int b, int x, int y)

{

return (a < x || a == x && b <= y);

}

inline bool leq(int a, int b, int c, int x, int y, int z)

{

return (a < x || a == x && leq(b, c, y, z));

}

inline int geti(int t, int nx, int sa[])

{

return (sa[t]<nx ? sa[t]\*3+1 : (sa[t]-nx)\*3+2);

}

static void radix(int a[], int b[], int s[], int n, int k)

{

int i, t, sum;

memset(ct, 0, (k + 1) \* sizeof(int));

for (i = 0; i < n; ++i) ct[s[a[i]]]++;

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

{

t = ct[i]; ct[i] = sum; sum += t;

}

for (i = 0; i < n; i++) b[ct[s[a[i]]]++] = a[i];

}

void suffix(int s[], int sa[], int n, int k)

{

int i, j, e, p, t;

int name = 0, cx = -1, cy = -1, cz = -1;

int nx = (n+2)/3, ny = (n+1)/3, nz = n/3, nxz = nx+nz;

int \*syz = s + n + 3, \*sayz = sa + n + 3;

for (i=0, j=0; i < n + (nx - ny); i++)

if (i%3 != 0) syz[j++] = i;

radix(syz , sayz, s+2, nxz, k);

radix(sayz, syz , s+1, nxz, k);

radix(syz , sayz, s , nxz, k);

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

{

if (s[ sayz[i] ] != cx || s[ sayz[i] + 1 ] != cy ||s[ sayz[i] + 2 ] != cz)

{

name++; cx = s[ sayz[i] ];

cy = s[ sayz[i] + 1 ]; cz = s[ sayz[i] + 2 ];

}

if (sayz[i] % 3 == 1) syz[ sayz[i] / 3 ] = name;

else syz[ sayz[i]/3 + nx ] = name;

}

if (name < nxz)

{

suffix(syz, sayz, nxz, name);

for (i = 0; i < nxz; i++) syz[sayz[i]] = i + 1;

}

else

{

for (i = 0; i < nxz; i++) sayz[syz[i] - 1] = i;

}

for (i = j = 0; i < nxz; i++)

if (sayz[i] < nx) sx[j++] = 3 \* sayz[i];

radix(sx, sax, s, nx, k);

for (p=0, t=nx-ny, e=0; e < n; e++)

{

i = geti(t, nx, sayz); j = sax[p];

if ( sayz[t] < nx ?leq(s[i], syz[sayz[t]+nx], s[j], syz[j/3]) :

leq(s[i], s[i+1], syz[sayz[t]-nx+1],

s[j], s[j+1], syz[j/3+nx]) )

{

sa[e] = i;

if (++t == nxz)

{

for (e++; p < nx; p++, e++)

sa[e] = sax[p];

}

}

else

{

sa[e] = j;

if (++p == nx) for (++e; t < nxz; ++t, ++e)

sa[e] = geti(t, nx, sayz);

}

}

}

void makesa()

{

memset(buf, 0, 4 \* n \* sizeof(int));

memset(sa, 0, 4 \* n \* sizeof(int));

for (int i=0; i<n; ++i) buf[i] = s[i] & 0xff;

suffix(buf, sa, n, 255);

}

void lcp()

{

int i, j, k;

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

while (k >= 0 && s[i] != s[ sa[j-1] + k ])

height[j] = (k--), j = rank[ sa[j] + 1 ];

}

void Rank()

{

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

rank[sa[i]]=i;

}

int main()

{

while(scanf("%s",s)!=EOF)

{

n=(int)strlen(s)+1;

makesa();

Rank();

lcp();

//……

}

return 0;

}

后缀自动机

//hdu 4622 多次询问一个字符串的某个子串的所有不相同子串

#define N 2010

int F[N\*2],ant,last,ch[N\*2][26],step[N\*2];

void init()

{

last=ant=1;

memset(F,0,sizeof(F));

memset(ch,0,sizeof(ch));

memset(step,0,sizeof(step));

}

void ins(int x)

{

int t=++ant,pa=last;

step[t]=step[last]+1;last=t;

for(;pa&&!ch[pa][x];pa=F[pa])

ch[pa][x]=t;

if(pa==0)

F[t]=1;

else if(step[pa]+1==step[ch[pa][x]])

F[t]=ch[pa][x];

else

{

int nq=++ant,q=ch[pa][x];

memcpy(ch[nq],ch[q],sizeof(ch[nq]));

step[nq]=step[pa]+1;F[nq]=F[q];F[q]=F[t]=nq;

for(;pa&&ch[pa][x]==q;pa=F[pa])

ch[pa][x]=nq;

}

}

//以上为后缀自动机的部分

char s[N];

int ans[10100];

vector<pair<int,int> > Q[N];

void run(int p)

{

int i,t,j,tt;

init();

sort(Q[p].begin(),Q[p].end());

for(i=0,t=p-1;i<(int)Q[p].size();i++)

{

if(i&&Q[p][i].first==t)

{

ans[Q[p][i].second]=tt;

continue;

}

while(t<Q[p][i].first)

{

ins(s[t]-'a');

t++;

}

for(tt=0,j=ant;j>0;j--)

tt+=step[j]-step[F[j]];

ans[Q[p][i].second]=tt;

}

}

int main()

{

int cas,n,i,a,b,m;

scanf("%d",&cas);

while(cas--)

{

scanf("%s",s);

m=(int)strlen(s);

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

Q[i].clear();

scanf("%d",&n);

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

{

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

Q[a].push\_back(make\_pair(b,i));

}

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

if(Q[i].size())

run(i);

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

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

}

return 0;

}

扩展KMP

/\*

\* 扩展KMP算法

\*/

//next[i]:x[i...m-1]与x[0...m-1]的最长公共前缀

//extend[i]:y[i...n-1]与x[0...m-1]的最长公共前缀

void pre\_EKMP(char x[],int m,int next[])

{

next[0]=m;

int j=0;

while(j+1 < m && x[j] == x[j+1])

j++;

next[1]=j;

int k=1;

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

{

int p=next[k]+k-1;

int L=next[i-k];

if(i+L < p+1)

next[i]=L;

else

{

j=max(0,p-i+1);

while(i+j < m && x[i+j] == x[j])

j++;

next[i]=j;

k=i;

}

}

}

//x是模式串，y是主串

void EKMP(char x[],int m,char y[],int n,int next[],int extend[])

{

pre\_EKMP(x,m,next);

int j=0;

while(j < n && j < m && x[j] == y[j])

j++;

extend[0]=j;

int k=0;

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

{

int p=extend[k]+k-1;

int L=next[i-k];

if(i+L < p+1)

extend[i]=L;

else

{

j=max(0,p-i+1);

while(i+j < n && j < m && y[i+j] == x[j])

j++;

extend[i]=j;

k=i;

}

}

}

//hdu4333

//题意：给定一个数(最大10^100000)，依次把最后一位数放到最前面，构成一个新的数，求有多少个数大于、小于和等于原来的数

//题解：先把原来数扩充一倍，然后根据扩展KMP比较(比较扩展KMP的下一位即可)。

//最后需要去重，只有原来的串有循环节的时候才会有重复串，用KMP的next数组求出最小循环节，用长度除以最小循环节得到循环节个数

char x[N\*2],y[N\*2];

int next[N\*2],extend[N\*2];

void pre\_EKMP(char x[],int m,int next[])

{

next[0]=m;

int j=0;

while(j+1 < m && x[j] == x[j+1])

j++;

next[1]=j;

int k=1;

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

{

int p=next[k]+k-1;

int L=next[i-k];

if(i+L < p+1)

next[i]=L;

else

{

j=max(0,p-i+1);

while(i+j < m && x[i+j] == x[j])

j++;

next[i]=j;

k=i;

}

}

}

void EKMP(char x[],int m,char y[],int n,int next[],int extend[])

{

pre\_EKMP(x,m,next);

int j=0;

while(j < n && j < m && x[j] == y[j])

j++;

extend[0]=j;

int k=0;

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

{

int p=extend[k]+k-1;

int L=next[i-k];

if(i+L < p+1)

extend[i]=L;

else

{

j=max(0,p-i+1);

while(i+j < n && j < m && y[i+j] == x[j])

j++;

extend[i]=j;

k=i;

}

}

}

void kmp\_pre(char x[],int len) {

int i, j;

j = next[0] = -1;

i = 0;

while (i < len) {

while (-1 != j && x[i] != x[j])

j = next[j];

next[++i] = ++j;

}

}

int main()

{

int T,cas=1;

sf(T);

while(T--)

{

printf("Case %d: ",cas++);

sfs(x);

int m=LEN(x);

int n=m\*2;

strcpy(y,x);

strcat(y,x);

EKMP(x,m,y,n,next,extend);

int s1=0,s2=0,s3=0;

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

{

if(extend[i] >= m)

s2++;

else if(y[i+extend[i]] < x[extend[i]])

s1++;

else

s3++;

}

kmp\_pre(x,m);

int tmp=m%(m-next[m])==0?m/(m-next[m]):1; //去重

printf("%d %d %d\n",s1/tmp,s2/tmp,s3/tmp);

}

return 0;

}

### 七、图论

tarjan求带重边无向图的割边

//N->边数，M->点数

const int N=100010;

const int M=10010;

struct edge {

int e,next,id;

int sign;

}ed[N];

int head[M],num;

int dfn[M],low[M],brg[N];

int brgnum,top,dp;

int n,m;

void add(int s,int e,int id)

{

ed[num].e=e;

ed[num].id=id;

ed[num].next=head[s];

ed[num].sign=0;

head[s]=num++;

}

void init()

{

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

num=0;

brgnum=0;

MEM(dfn);

MEM(low);

MEM(brg);

top=dp=0;

}

void tarjan(int now,int fa)

{

dfn[now]=low[now]=++dp;

for(int i=head[now];~i;i=ed[i].next)

{

int e=ed[i].e;

if(ed[i].sign) continue;

ed[i].sign=ed[i^1].sign=1;

if(!dfn[e])

{

tarjan(e,now);

low[now]=min(low[now],low[e]);

if(dfn[now] < low[e])

brg[brgnum++]=ed[i].id;

}

else

low[now]=min(low[now],dfn[e]);

}

}

void solve()

{

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

{

top=dp=0;

if(!dfn[i])

tarjan(i,-1);

}

sort(brg,brg+brgnum);

pf(brgnum);

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

{

if(i) pfk;

printf("%d",brg[i]);

}

pfn;

}

int main()

{

while(sf2(n,m)!=EOF)

{

init();

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

{

int a,b;

sf2(a,b);

add(a,b,i);

add(b,a,i);

}

solve();

}

return 0;

}

并查集

#define N 10010

int f[N];

//结点标号从1到n，若有不同自行改正

void init(int n)

{

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

f[i]=i;

}

int Find(int a)

{

if(a != f[a])

f[a]=Find(f[a]);

return f[a];

}

void Merge(int x,int y)

{

int fx,fy;

fx=Find(x);

fy=Find(y);

if(fx != fy)

f[fx]=fy;

}

int main()

{

int n,m,a,b,i,res;

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

{

init(n);

while(m--)

{

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

Merge(a,b);

}

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

if(i == f[i])

res++;

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

}

return 0;

}

最大团

问题描述：团就是最大完全子图。

给定无向图G=(V,E)。如果U属于V，且对任意u，v属于U，有(u，v)属于E，则称U是G的完全子图。

G 的完全子图U是G的团当且仅当U不包含在G的更大的完全子图中，即U就是最大完全子图。

G 的最大团是指G中所含顶点数最多的团。

//dp[i]:从i到n-1的最大的团

//mx最后的结果

//stk[i][j]:第i层中与之相连的第j大的标号

//g[][]是邻接矩阵

//res=cilque(n);

int g[N][N],dp[N],stk[N][N],mx;

//总共有n个数,dep代表当前的层数,ns代表于当前层相连的并且比ns大的标号的个数

int dfs(int n,int ns,int dep)

{

if(ns == 0)

{

if(dep > mx)

mx=dep;

return 1;

}

int k,p,cnt;

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

{

k=stk[dep][i];//与之相连的第i个点

if(dep+n-k <= mx)//当前层数+第k层下边的<=mx,则不再搜索

return 0;

if(dep+dp[k] <= mx)//当前层数+dp的最大的<=mx,不再搜索

return 0;

cnt=0;

for(int j=i+1;j<ns;j++)

{

p=stk[dep][j];//i后边的某个点

if(g[k][p])//如果i和j相连

stk[dep+1][cnt++]=p;//如果没有与之相连的,则cnt为0

}

dfs(n,cnt,dep+1);

}

return 1;

}

int clique(int n)

{

int ns;

mx=0;

//dp用的,vertex: 0 ~ n-1

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

{

ns=0;

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

if(g[i][j])

stk[1][ns++]=j;

dfs(n,ns,1);

dp[i]=mx;

}

return mx;

}

最小生成树

点少选择Prim算法O(n^2)，边少选择kruskal算法O(eloge)

//Prim算法

#define N 1020

#define M 0x3fffffff

int n,f[N][N];

int lowcost[N];

bool vis[N];

int MST\_Prim()

{

//设0为起点

int i,j,sum,minnum,sign;

sum=0;

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

lowcost[i]=f[0][i];

memset(vis,false,sizeof(vis));

vis[0]=true;

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

{

minnum=M;

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

if(!vis[j] && lowcost[j]<minnum)

{

sign=j;

minnum=lowcost[j];

}

if(minnum==M)

break;

sum+=minnum;

vis[sign]=true;

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

if(!vis[j] && f[sign][j]<lowcost[j])

lowcost[j]=f[sign][j];

}

return sum;

}

//kruskal算法

#define N 10020

#define M 0x3fffffff

struct Node{

int s,e,v;

}node[N];

int f[N];

bool cmp(Node a,Node b)

{

if(a.v < b.v)

return true;

return false;

}

int Find(int a)

{

if(a != f[a])

f[a]=Find(f[a]);

return f[a];

}

//n个点，m条边

int kruskal(int n,int m)

{

int fx,fy,sum=0;

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

f[i]=i;

sort(node,node+m,cmp);

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

{

fx=Find(node[i].s);

fy=Find(node[i].e);

if(fx != fy)

{

sum+=node[i].v;

f[fx]=f[fy];

}

}

return sum;

}

最短路-Dijkstra

const int maxnum = 110;

const int maxint = 999999;

int dist[maxnum];

int c[maxnum][maxnum];

int n, line;

void Dijkstra(int v)

{

int i,j,tmp,u;

bool s[maxnum];

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

{

dist[i] = c[v][i];

s[i] = 0;

}

dist[v] = 0;

s[v] = 1;

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

{

tmp = maxint;

u = v;

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

if((!s[j]) && dist[j]<tmp)

{

u = j;

tmp = dist[j];

}

if(tmp==maxint)

return ;

s[u] = 1;

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

if((!s[j]) && dist[u] + c[u][j] < dist[j])

dist[j]=dist[u] + c[u][j];

}

}

int main()

{

int p, q, len,i,j;

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

{

if(n==0 && line==0)

break;

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

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

c[i][j] = maxint;

while(line--)

{

scanf("%d%d%d",&p,&q,&len);

if(len < c[p][q])

{

c[p][q] = len;

c[q][p] = len;

}

}

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

dist[i] = maxint;

Dijkstra(1);

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

}

return 0;

}

最短路-Bellman-Ford

const int N = 205;

const int M = 20005;

const int MAX = 1000000000;

int dis[N];

struct edge

{

int u;

int v;

int w;

}e[M];

void init(int vs, int s)

{

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

dis[i] = MAX;

dis[s] = 0;

return ;

}

bool relax(int u, int v, int w)

{

if (dis[v] > dis[u] + w)

{

dis[v] = dis[u] + w;

return true;

}

return false;

}

void bellmanFord(int es, int vs, int s)

{

//es表示边数，vs表示点数，s表示起点

init(vs, s);

int flag;

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

{

flag=false;

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

if(relax(e[j].u, e[j].v, e[j].w))

flag=true;

if(!flag)

return ;

}

}

int main()

{

int n, m;

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

{

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

{

scanf ("%d%d%d", &e[i].u, &e[i].v, &e[i].w);

e[m+i].u = e[i].v;

e[m+i].v = e[i].u;

e[m+i].w = e[i].w;

}

bellmanFord(m<<1, n, 1);

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

}

return 0;

}

最短路-Floyed

#define inf 0x3f3f3f3f

int n,m,p,q;

int f[110][110];

void floyed()

{

int i,j,k;

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

{

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

{

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

{

if(f[i][k]+f[k][j]<f[i][j])

f[i][j]=f[i][k]+f[k][j];

}

}

}

if(f[1][n]==inf)

printf("0\n");

else

printf("%d\n",f[1][n]);

}

int main()

{

int i,a,b,c;

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

{

if(n==0 && m==0)

break;

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

{

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

f[p][q]=inf;

}

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

{

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

f[a][b]=c;

f[b][a]=c;

}

floyed();

}

return 0;

}

最短路-SPFA

#define MAX 999999999

#define N 1010

int dis[N],n,m;

int edge[N][N];

bool vis[N];

void SPFA(int s)

{

int i,u;

memset(vis,false,sizeof(vis));

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

dis[i]=MAX;

queue<int>Q;

Q.push(s);

vis[s]=true;

dis[s]=0;

while(!Q.empty())

{

u=Q.front();

Q.pop();

vis[u]=false;

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

{

if(dis[i]>dis[u]+edge[u][i])

{

dis[i]=dis[u]+edge[u][i];

if(!vis[i])

{

vis[i]=true;

Q.push(i);

}

}

}

}

}

int main()

{

int i,j,a,b,c;

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

{

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

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

{

edge[i][j]=MAX;

edge[j][i]=MAX;

}

}

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

{

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

if(edge[a][b]>c)

{

edge[a][b]=c;

edge[b][a]=c;

}

}

SPFA(1);

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

}

return 0;

}

欧拉回路

struct Edge{

int fro,to,next,val;

}edge[N];

int n,tot,res;

int path[N],vis[N],head[N];

//head数组记录一个点对应多个点的边信息，类似vector

void add(int fro,int to,int val)

{

edge[tot].val=val;

edge[tot].fro=fro;

edge[tot].to=to;

edge[tot].next=head[fro];

head[fro]=tot++;

}

void dfs(int t)

{

for(int i=head[t];i!=-1;i=edge[i].next)

if(!vis[edge[i].val])

{

vis[edge[i].val]=1;

dfs(edge[i].to);

path[res++]=edge[i].val; //这里path记录边的权值

}

}

void solve()

{

tot=0;res=0;

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

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

{

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

add(a,b,c);

}

MEM(vis);

dfs(0);

for(int i=res-1;i>=0;i--) //倒序输出

printf("%d",path[i]);

printf("\n");

}

拓扑排序

定义：

有n个变量，m个二元组（u,v），分别表示变量u小于v，求最后的排序结果。（当然这里二元组的关系可以有多种解释，有时候可以理解为先后顺序，即一个大任务分解成若干个小任务，必须先完成一些小任务后才能完成一个任务）

分析：

把每个变量看成一个点，二元组的关系看成有向边，这样就能得到一个有向图。这样，我们的任务实际上是把一个图的所有结点排序，使得每一个有向边(u,v)对应的u都排在v的前面。当然图中不可以出现环。

实现方法：

1、选择有向图中一个没有前驱的结点并输出，即该结点的入度为1

2、在有向图中删除该点，即删除该点出发的所有边

3、重复上述两步，直到输出所有点

//hdu1285 有n个队伍比赛，m条比赛结果，即a赢b，求最后的排名。

#define N 520

int n,g[N][N];

int ru[N],vis[N],res[N];

//res记录排序后的结果,ru数组记录各个点的入度之和，vis数组记录该点是否被访问

void Top\_sort()

{

int i,j,k;

memset(ru,0,sizeof(ru));

memset(vis,0,sizeof(vis));

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

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

if(g[i][j]==1)

ru[j]++;

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

{

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

{

if(!vis[j] && ru[j]==0)

{

res[i]=j;

vis[j]=1;

//更新，把从j出发的边去掉，即相应的入度减一

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

if(g[j][k]==1)

ru[k]--;

break;

}

}

// if(j-1 == n) //如果成立则表示有环存在，看题目要求处理

// return ;

}

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

{

if(i!=1)

printf(" ");

printf("%d",res[i]);

}

printf("\n");

}

int main()

{

int m,a,b;

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

{

memset(g,0,sizeof(g));

while(m--)

{

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

g[a][b]=1;

}

Top\_sort();

}

return 0;

}

### 八、网络流

二分图最大匹配-匈牙利算法

//n表示左边集合元素个数，m表示右边集合元素个数

int n, m, mat[N];

bool vis[N],f[N][N];

bool dfs(int k)

{

int t;

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

{

if (f[k][i] && !vis[i])

{

vis[i]=true;

t=mat[i];

mat[i]=k;

if (t==-1 || dfs(t))

return true;

mat[i]=t;

}

}

return false;

}

int hungary()

{

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

int ans=0;

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

{

memset(vis,false,sizeof(vis));

if (dfs(i))

ans++;

}

return ans;

}

完备匹配下最大权匹配-KM算法

KM算法求的是完备匹配下的最大权匹配： 在一个二分图内，左顶点为X，右顶点为Y，现对于每组左右连接XiYj有权wij，求一种匹配使得所有wij的和最大。

/\*

\* 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。res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res

\*/

int w[N][N];

int lx[N], ly[N];

int linky[N];

int visx[N], visy[N];

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;

if(linky[i] != -1)

res += w[linky[i]][i];

}

return true;

}

稳定婚姻问题

【稳定婚姻问题】

问题来自于一场“3分钟相亲”活动，参加活动的有n位男士和n位女士。要求每位男士都要和所有的女士进行短暂的单独交流，并为她们打分，然后按照喜欢程度，对每一位女士进行排序；同样的，每位女士也要对所有男士进行打分和排序。

作为活动的组织者，当你拿到这些数据后，该如何为男，女士们配对，才能使大家皆大欢喜，组成稳定的婚姻呢？即：满足“除妻子（丈夫）外，我爱的人不爱我，爱我的人我不爱”条件。

1962 年，美国数学家 David Gale 和 Lloyd Shapley 发明了一种寻找稳定婚姻的策略，人们称之为延迟认可算法（Gale-Shapley算法）。

先对所有男士进行落选标记，称其为自由男。当存在自由男时，进行以下操作：

① 每一位自由男在所有尚未拒绝她的女士中选择一位被他排名最优先的女士；

② 每一位女士将正在追求她的自由男与其当前男友进行比较，选择其中排名优先的男士作为其男友，即若自由男优于当前男友，则抛弃前男友；否则保留其男友，拒绝自由男。

③ 若某男士被其女友抛弃，重新变成自由男。

在算法执行期间，自由男们主动出击，依次对最喜欢和次喜欢的女人求爱，一旦被接受，即失去自由身，进入订婚状态；而女人们则采取“守株待兔”和“喜新厌旧”策略，对前来求爱的男士进行选择：若该男子比未婚夫强，则悔婚，选择新的未婚夫；否则拒绝该男子的求婚。被女友抛弃的男人重获自由身，重新拥有了追求女人的权利——当然，新的追求对象比不过前女友。

这样，在算法执行期间，每个人都有可能订婚多次——也有可能一开始就找到了自己的最爱，从一而终——每订一次婚，女人们的选择就会更有利，而男人们的品味则越来越差。只要男女生的数量相等，则经过多轮求婚，订婚，悔婚和再订婚之后，每位男女最终都会找到合适的伴侣——虽然不一定是自己的最爱（男人没能追到自己的最爱，或女人没有等到自己的最爱来追求），但绝对不会出现“虽然彼此相爱，却不能在一起”的悲剧，所有人都会组成稳定的婚姻。

代码如下：

int libMan[N][N],libLady[N][N+1]; //男士和女士所喜欢的异性序号排列表

int n,man[N+1],lady[N];

bool ChangeFriend(int v,int oldF,int newF) //判断是否需要换男友

{

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

{

if(libLady[v][i] == oldF)

{

oldF=i;

break;

}

}

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

{

if(libLady[v][i] == newF)

{

newF=i;

break;

}

}

return oldF > newF;

}

void Gale\_Shapley()

{

memset(man,0,sizeof(man));

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

lady[i]=n;

int i=0;

while(i < n) //第一个男士开始选

{

int v=libMan[i][man[i]]; //第i个男士喜欢第v个女士

if(i == lady[v]) //i号男就是v号女当前男友，跳过

i++;

else if(ChangeFriend(v,lady[v],i)) //如果i号男比v号女当前男友优秀，则v抛弃前男友，重新选择

{

int t=lady[v]; //t存储前男友序号

man[lady[v]]++; //抛弃前男友，前男友选择其“次喜欢女”

lady[v]=i; //选择i号男为新男友

if(t > i) //前男友t在新男友i之后，则处理下一个男士

i++;

else //否则先处理前男友t

i=t;

}

else //继续处理i号男的“次喜欢女”

man[i]++;

}

}

void work()

{

//输出每位男士追求女士的次数

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

printf("%d ",man[i]+1);

printf("\n");

//输出每位男士的妻子序号

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

printf("%d ",libMan[i][man[i]]);

printf("\n");

//输出每位女士的丈夫序号

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

printf("%d ",lady[i]);

printf("\n");

}

### 九、树论

LCA离线Tarjan算法

LCA离线Tarjan算法：

存储树的时候存储的是单向边：

伪代码如下：

Tarjan(u)

fa[u] = u

循环u的每个孩子v

if vis[v] == 0

Tarjan(v)

fa[v] = u;

vis[u] = 1

循环u相关的每个询问(u,v)

if vis[v] == 1

求u和v的LCA为 Find(v)

vis[u] = 1 写在中间不会出现死循环，这样可以避免重复计算LCA

存储树的时候存储的是双向边：

伪代码如下：

Tarjan(u)

vis[u] = 1

fa[u] = u

循环u的每个孩子v

if vis[v] == 0

Tarjan(v)

fa[v] = u;

循环u相关的每个询问(u,v)

if vis[v] == 1

求u和v的LCA为 Find(v)

vis[u] = 1 写在中间不会出现死循环，这样可以避免重复计算LCA

另外，可以用两个不同的标记数组标记访问次序可以完全解决这个问题。

LCA离线Tarjan算法-hdu2586

/\*

\* hdu 2586

\* n 个点，n-1 条有权值边，q 个询问，问任意两点的距离

\*/

struct Edge{

int v,w,next;

}edge[N\*2];

struct Node{

int id,to; //id 记录第几次询问

Node(int x\_id,int x\_to){

id=x\_id;

to=x\_to;

}

};

int n,m;

int e,head[N];

int vis[N],fa[N],dis[N],ans[N]; //dis[] 记录树根到点距离，ans[] 记录询问结果

vector<Node>que[N]; //记录询问

void init() {

e=0;

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

head[i]=-1;

vis[i]=0;

dis[i]=0;

fa[i]=i;

que[i].clear();

}

}

void add(int u,int v,int w){

edge[e].v=v;

edge[e].w=w;

edge[e].next=head[u];

head[u]=e++;

}

int Find(int x) {

return x==fa[x]?x:fa[x]=Find(fa[x]);

}

void Merge(int x,int y) {

int fx=Find(x);

int fy=Find(y);

if(fx != fy){

fa[fy]=fx;

}

}

/\*

\* 当查询到当前点 u 时，则 u 和其子树的 lca 就是 u，用并查集合并

\* 同理，u 的父结点和 u 的兄弟结点的 lca 就是 u 的父结点了

\* 用同样方法更新子树

\*/

void Tarjan(int u,int step) {

dis[u]=step;

vis[u]=1;

for(int i=head[u];i!=-1;i=edge[i].next){

int t=edge[i].v;

if(vis[t]) continue;

Tarjan(t,step+edge[i].w);

Merge(u,t);

}

//计算与 u 相关的查询

int tot=(int)que[u].size();

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

int t=que[u][i].to;

if(!vis[t]) continue;

ans[que[u][i].id]=dis[u]+dis[t]-2\*dis[Find(t)]; //两点距离等于根到两点距离和减 2 倍根到 lca 距离

}

}

void solve(){

int a,b,c;

sf2(n,m);

init();

fr(n-1) {

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

add(a,b,c);

add(b,a,c);

}

fr(m){

sf2(a,b);

que[a].push\_back(Node(i,b));

que[b].push\_back(Node(i,a));

}

Tarjan(1,0);

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

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

}

}

LCA离线Tarjan算法-poj1470

/\* poj 1470

\* 给 n 个点的树，q 次查询，每次询问两个点的 LCA，问最后每个点成为 LCA 的次数

\* 注意：这里存单边统计方便，不会有重复统计

\*/

struct Edge{

int v,next;

}edge[N\*2];

int n,m;

int e,head[N];

int vis[N],fa[N],r[N],gen[N];

vector<int>que[N];

void init() {

e=0;

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

head[i]=-1;

vis[i]=0;

fa[i]=i;

que[i].clear();

r[i]=0;

gen[i]=0;

}

}

void add(int u,int v){

edge[e].v=v;

edge[e].next=head[u];

head[u]=e++;

}

int Find(int x) {

return x==fa[x]?x:fa[x]=Find(fa[x]);

}

void Merge(int x,int y) {

int fx=Find(x);

int fy=Find(y);

if(fx != fy){

fa[fy]=fx;

}

}

void Tarjan(int u) {

for(int i=head[u];i!=-1;i=edge[i].next){

int t=edge[i].v;

if(vis[t]) continue;

Tarjan(t);

Merge(u,t);

}

vis[u]=1; //存的是单边，这句如果放前面，会重复统计

int tot=(int)que[u].size();

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

int t=que[u][i];

if(!vis[t]) continue;

r[Find(t)]++;

}

}

int get\_int()

{

int ch,res=0;

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;

}

void solve(){

int a,b,t;

init();

fr(n) {

a=get\_int();

t=get\_int();

while(t--) {

scanf("%d",&b);

gen[b]=1;

add(a,b); //存单边

}

}

sf(m);

fr(m){

a=get\_int();

b=get\_int();

que[a].push\_back(b); //两个点相关的查询都记录

que[b].push\_back(a);

}

for(int i=1;i<=n;i++){ //确定根结点

if(!gen[i]){

Tarjan(i);

break;

}

}

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

if(r[i]){

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

}

}

}

int main()

{

while(sf(n)!=EOF){

solve();

}

return 0;

}

LCA在线算法

struct edge{

int u,v,w,next;

}edge[2\*N];

int n,q;

int tot,e,head[N];

int vis[N],dp[2\*N][M];

int ver[2\*N],R[2\*N],first[N],dir[N];

void init(){

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

memset(vis,0,sizeof(vis));

num = 0;tot = 0; dir[1] = 0;

}

inline void add(int u ,int v ,int w){

edge[e].u = u;

edge[e].v = v;

edge[e].w = w;

edge[e].next = head[u];

head[u] = e++;

}

void dfs(int u ,int dep){

vis[u] = 1; ver[++tot] = u; first[u] = tot; R[tot] = dep;

for(int k=head[u]; k!=-1; k=edge[k].next)

if( !vis[edge[k].v] ){

int v = edge[k].v , w = edge[k].w;

dir[v] = dir[u] + w;

dfs(v,dep+1);

ver[++tot] = u; R[tot] = dep;

}

}

void ST(int n){

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

dp[i][0] = i;

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

{

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

{

int a = dp[i][j-1] , b = dp[i+(1<<(j-1))][j-1];

dp[i][j] = R[a]<R[b]?a:b;

}

}

}

int RMQ(int l,int r)

{

int k=0;

while((1<<(k+1))<=r-l+1)

k++;

int a = dp[l][k], b = dp[r-(1<<k)+1][k];

return R[a]<R[b]?a:b;

}

int LCA(int u ,int v)

{

int x = first[u] , y = first[v];

if(x > y) swap(x,y);

int res = RMQ(x,y);

return ver[res];

}

void solve(){

init();

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

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

{

int u,v,w;

scanf("%d%d%d",&u,&v,&w);

add(u,v,w);

add(v,u,w);

}

dfs(1,1);

ST(2\*n-1);

while(q--)

{

int u,v;

scanf("%d%d",&u,&v);

int lca = LCA(u,v);

printf("%d\n",dir[u] + dir[v] - 2\*dir[lca]);

}

}

求哈夫曼树的WPL

给N个权值Wi构成的有N个叶子结点的哈夫曼树，相应的叶子结点的路径长度为Li，求哈夫曼树的WPL

priority\_queue <LL, vector<LL>, greater<LL> > Q;

int main()

{

int num;

LL n,m,a;

while(sf(num)!=EOF)

{

if(num == 1) //特判

{

sf64(a);

pf64(a);

continue ;

}

while(!Q.empty())

Q.pop();

fr(num){

sf64(a);

Q.push(a);

}

LL sum=0;

while(Q.size() > 1)

{

n=Q.top();Q.pop();

m=Q.top();Q.pop();

sum=sum+n+m;

Q.push(n+m);

}

pf64(sum);

}

return 0;

}

求树的直径

//边带权值的无向联通图

//u表示直径端点，ans表示树的直径d

//无向图边大小开两倍

struct Edge{

int v,w,next;

}edge[N\*2];

int n,m;

int e,head[N];

int ans,u;

void init(){

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

e=0;

}

void add(int u,int v,int w){

edge[e].v=v;

edge[e].w=w;

edge[e].next=head[u];

head[u]=e++;

}

void dfs(int tmp,int pre,int step) {

if(step > ans){

ans=step;

u=tmp;

}

for(int i=head[tmp];i!=-1;i=edge[i].next){

if(edge[i].v != pre) //保证往下级搜索

dfs(edge[i].v,tmp,step+edge[i].w);

}

}

void solve() {

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

init();

int a,b,c;

fr(m){

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

add(a,b,c);

add(b,a,c);

}

ans=-1;

dfs(1,-1,0);

ans=-1;

dfs(u,-1,0);

pf(ans);

}

求树中每个结点的子树结点个数

//一共n个结点，n-1条边，边信息存在容器里，f[]数组记录每个结点的子树结点个数，fat[]数组记录每个结点的父结点

int n,f[N];

int vis[N],fat[N];

vector<int> s[N];

int dfs(int tmp)

{

int res=1;

for(int i=0;i<(int)s[tmp].size();i++)

{

int a=s[tmp][i];

if(!vis[a])

{

vis[a]=1;fat[a]=tmp;

res=res+dfs(a);

}

}

f[tmp]=res;

return res;

}

void solve()

{

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

s[i].clear();

int a,b;

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

{

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

s[a].push\_back(b);

s[b].push\_back(a);

}

MEM(vis);vis[1]=1;fat[1]=0;

a=dfs(1);

}

已知二叉树先序遍历和中序遍历

求后序遍历

typedef struct BiTNode

{

int data,temp;

struct BiTNode \*lchild,\*rchild;

}BiTNode,\*BiTree;

int n,flag,f1[N],f2[N];

void dfs(BiTree &root,int t)

{

if(root == NULL)

{

root=new BiTNode;

root->data=f2[t];

root->temp=t;

root->lchild=NULL;

root->rchild=NULL;

return ;

}

if(t < root->temp)

dfs(root->lchild,t);

else

dfs(root->rchild,t);

}

void Create(BiTree &root)

{

int tmp;

root=new BiTNode;

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

if(f2[i] == f1[0])

{

tmp=i;

root->data=f1[0];

root->temp=i;

root->lchild=NULL;

root->rchild=NULL;

break;

}

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

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

{

if(f1[i] == f2[j])

{

if(j < tmp)

dfs(root->lchild,j);

else

dfs(root->rchild,j);

break;

}

}

}

void Output(BiTree root)

{

if(root == NULL)

return ;

Output(root->lchild);

Output(root->rchild);

if(flag == 1)

printf(" ");

printf("%d",root->data);

flag=1;

}

void solve()

{

scanf("%d",&n);

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

scanf("%d",&f1[i]);

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

scanf("%d",&f2[i]);

BiTree root;

Create(root);

flag=0;

Output(root);

printf("\n");

}

### 十、数据结构

RMQ

//arr是目标数组，n表示长度，求区间内最小（大）值，数组下标从1开始

int rmq\_n[N][32];

void rmq\_init(int arr[],int n)

{

int i,j;

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

rmq\_n[i][0]=arr[i];//rmq\_m[i][0]=arr[i];

for(j=1;j<=int(log(n+0.0)/log(2.0));++j)

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

rmq\_n[i][j]=min(rmq\_n[i][j-1],rmq\_n[i+(1<<(j-1))][j-1]);

//rmq\_m[i][j]=max(rmq\_m[i][j-1],rmq\_m[i+(1<<(j-1))][j-1]);

}

int rmq(int a,int b)

{

int m=int(log(b-a+1.0)/log(2.0));

int x=min(rmq\_n[a][m],rmq\_n[b-(1<<m)+1][m]);

//int x=max(rmq\_m[a][m],rmq\_m[b-(1<<m)+1][m]);

return x;

}

//求区间内最小（大）值的数组小标

int rmq\_n[N][32];

int check(int i,int j)

{

return f[i]<=f[j]?i:j;

}

void rmq\_init(int arr[],int n)

{

int i,j;

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

rmq\_n[i][0]=i;

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

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

rmq\_n[i][j]=check(rmq\_n[i][j-1],rmq\_n[i+(1<<(j-1))][j-1]);

}

int rmq(int a,int b)

{

int m=int(log(b-a+1.0)/log(2.0));

int x=check(rmq\_n[a][m],rmq\_n[b-(1<<m)+1][m]);

return x;

}

树状数组

树状数组：

用数组d[i]记录i点管辖范围内的和，则有：

d[1]=a[1];

d[2]=a[1]+a[2];

d[3]=a[3];

d[4]=a[1]+a[2]+a[3]+a[4];

d[5]=a[5];

d[6]=a[5]+a[6];

d[7]=a[7];

d[8]=a[1]+a[2]+a[3]+a[4]+a[5]+a[6]+a[7]+a[8];

……

可以发现第i个点管辖的数有2^k个，k表示i的二进制数从个位往高位看0的个数。

这样的构成的树，适用于：

对一个数组内部动态的修改（加，减，乘，除），删除，增加，或者高效求某个点或者某个区间的值；时间复杂度均为O(logN)。

//求某个点的管辖范围

int lowbit(int x)

{

return x&(-x);

}

用空间代替时间：

int Lowbit[N];

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

Lowbit[i] = i & (i ^ (i - 1));

/\*

求数组的和的算法

（1）首先，令sum=0，转向第二步；

（2）接下来判断，如果 n>0 的话，就令sum=sum+cn转向第三步，否则的话，终止算法，返回 sum 的值；

（3）n=n - lowbit（n）（将n的二进制表示的最后一个零删掉），回第二步。

\*/

代码如下：

int Sum(int n)

{

int sum=0;

while(n>0)

{

sum+=c[n];

n=n-lowbit(n);

}

return sum;

}

/\*

对数组中的元素做修改，算法如下（修改为给某个节点 i 加上 x ）：

（1）当 i<=n 时，执行下一步；否则的话，算法结束;

（2）ci=ci+x ，i=i+lowbit（i）（在 i 的二进制表示的最后加零），返回第一步。

\*/

代码如下：

void Update(int i,int x)

{

while(i<=n)

{

c[i]=c[i]+x;

i=i+lowbit(i);

}

}

树状数组解区间更新 询问值问题

//hdu 1556

#define N 100010

int ans[N];

int n;

int lowBit(int x)

{

return x&-x;

}

int Sum(int x)

{

int sum=0;

while(x > 0)

{

sum+=ans[x];

x-=lowBit(x);

}

return sum;

}

void Update(int x,int y)

{

while(x<=n)

{

ans[x]+=y;

x+=lowBit(x);

}

}

int main()

{

int a,b;

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

{

memset(ans,0,sizeof(ans));

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

{

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

Update(a,1);//先更新a到n之间的值加1

Update(b+1,-1);//再更新b+1到n之间的值减1

}

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

{

if(i!=1)

printf(" ");

printf("%d",Sum(i));//第i项值为1到i的和

}

printf("\n");

}

return 0;

}

线段树

struct Node{

int left,right,v;

// int mn,mx; //此处根据要求设定变量

}Tree[N\*3]; //N大小设为点个数三倍左右

void Build(int id,int left,int right) //建树

{

Tree[id].v=0;

Tree[id].left=left;

Tree[id].right=right;

if(left == right) //此处设叶子节点初始值

return;

int mid=(left+right)>>1;

Build(id<<1,left,mid);

Build(id<<1|1,mid+1,right);

// Tree[id].v=min(Tree[id<<1].v,Tree[id<<1|1].v); //此处根据要求更新Tree[id]的最值或者和

}

void Update(int id,int pos,int add) //单点更新，pos位加上add

{

int left=Tree[id].left,right=Tree[id].right;

if(left == right){

Tree[id].v+=add;

return;

}

int mid=(left+right)>>1;

if(mid>=pos)

Update(id<<1,pos,add);

else

Update(id<<1|1,pos,add);

// Tree[id].v=min(Tree[id<<1].v,Tree[id<<1|1].v); //此处根据要求更新Tree[id]的最值或者和

}

//Insert函数和Update差不多，

void Insert(int id,int pos,int num) //pos位值设为num

{

int left=Tree[id].left,right=Tree[id].right;

if(left == right)

{

Tree[id].v=num;

return;

}

int mid=(left+right)>>1;

if(mid >= pos)

Insert(id<<1,pos,num);

else

Insert(id<<1|1,pos,num);

// Tree[id].mn=min(Tree[id<<1].mn,Tree[id<<1|1].mn); //此处根据要求更新Tree[id]的最值或者和

}

int Query(int id,int Qleft,int Qright) //查询区间最值或者和

{

int left=Tree[id].left,right=Tree[id].right;

if(left >= Qleft && right <= Qright)

return Tree[id].v; //也可改成void类型函数，开全局变量更新，要学会灵活运用

int mid=(left+right)>>1;

if(mid >= Qright)

return Query(id<<1,Qleft,Qright);

else if(mid < Qleft)

return Query(id<<1|1,Qleft,Qright);

else

{

int r1=Query(id<<1,Qleft,mid);

int r2=Query(id<<1|1,mid+1,Qright);

// return min(r1,r2); //此处根据要求返回最值或者和

}

}

void solve()

{

int n,m,k,a,b;

sf3(n,k,m); //n个点，k个更新，m个查询

Build(1,1,n);

for(int i=1;i<=n;i++) //输入初始值

{

sf(a);

Insert(1,i,a); //插入初始值

}

while(k--)

{

sf2(a,b);

Update(1,a,b); //a位置上的值加上b

}

while(m--)

{

sf2(a,b);

pf(Query(1,a,b)); //区间[a,b]的最值或者和

}

}

/\*

\* 关于线段树区间更新，不需要更新到叶子结点，只需更新到相应的线段。

\* 设记录变量tmp，更新或者查询其子结点的时候，将tmp往下更新，更新其子节点的和以及tmp，然后把当前的结点的tmp设为0

\* 典型例题：poj3468 给n个数，m个操作，操作一：把区间[a,b]内所有值加上c；操作二：求区间[a,b]和

\*/

struct Node{

int left,right;

LL tmp,s;

}Tree[N\*3]; //N大小设为点个数三倍左右

int ans;

void Build(int id,int left,int right) //建树

{

Tree[id].tmp=0;

Tree[id].left=left;

Tree[id].right=right;

if(left == right)

{

sf64(Tree[id].s);

return;

}

int mid=(left+right)>>1;

Build(id<<1,left,mid);

Build(id<<1|1,mid+1,right);

Tree[id].s=Tree[id<<1].s+Tree[id<<1|1].s;

}

void Update(int id,int add,int Qleft,int Qright)

{

int left=Tree[id].left,right=Tree[id].right;

if(left >= Qleft && right <= Qright)

{

Tree[id].s+=(Tree[id].right-Tree[id].left+1)\*add;

Tree[id].tmp+=add;

return ;

}

if(Tree[id].tmp) //如果当前结点tmp不为空，则将tmp往下更新

{

Tree[id<<1].s+=(Tree[id<<1].right-Tree[id<<1].left+1)\*Tree[id].tmp;

Tree[id<<1|1].s+=(Tree[id<<1|1].right-Tree[id<<1|1].left+1)\*Tree[id].tmp;

Tree[id<<1].tmp+=Tree[id].tmp;

Tree[id<<1|1].tmp+=Tree[id].tmp;

Tree[id].tmp=0;

}

int mid=(left+right)>>1;

if(mid >= Qright)

Update(id<<1,add,Qleft,Qright);

else if(mid < Qleft)

Update(id<<1|1,add,Qleft,Qright);

else

{

Update(id<<1,add,Qleft,mid);

Update(id<<1|1,add,mid+1,Qright);

}

Tree[id].s=Tree[id<<1].s+Tree[id<<1|1].s;

}

LL Query(int id,int Qleft,int Qright)

{

int left=Tree[id].left,right=Tree[id].right;

if(left >= Qleft && right <= Qright)

return Tree[id].s;

if(Tree[id].tmp) //如果当前结点tmp不为空，则将tmp往下更新

{

Tree[id<<1].s+=(Tree[id<<1].right-Tree[id<<1].left+1)\*Tree[id].tmp;

Tree[id<<1|1].s+=(Tree[id<<1|1].right-Tree[id<<1|1].left+1)\*Tree[id].tmp;

Tree[id<<1].tmp+=Tree[id].tmp;

Tree[id<<1|1].tmp+=Tree[id].tmp;

Tree[id].tmp=0;

}

int mid=(left+right)>>1;

if(mid >= Qright)

return Query(id<<1,Qleft,Qright);

else if(mid < Qleft)

return Query(id<<1|1,Qleft,Qright);

else

{

LL r1=Query(id<<1,Qleft,mid);

LL r2=Query(id<<1|1,mid+1,Qright);

return r1+r2;

}

}

int main()

{

char ss[5];

int n,m,a,b,c;

while(sf2(n,m)!=EOF && n)

{

Build(1,1,n);

while(m--)

{

sfs(ss);

if(ss[0] == 'Q')

{

sf2(a,b);

pf64(Query(1,a,b));

}

else

{

sf3(a,b,c);

Update(1,c,a,b);

}

}

}

return 0;

}

### 十一、动态规划

背包九讲

P01: 01背包问题

题目

有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。用子问题定义状态：即f[i][v]表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。则其状态转移方程便是：f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前i件物品放入容量为v的背包中”这个子问题，若只考虑第i件物品的策略（放或不放），那么就可以转化为一个只牵扯前i-1件物品的问题。如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”；如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-c[i]的背包中”，此时能获得的最大价值就是f [i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]。

注意f[i][v]有意义当且仅当存在一个前i件物品的子集，其费用总和为v。所以按照这个方程递推完毕后，最终的答案并不一定是f[N] [V]，而是f[N][0..V]的最大值。如果将状态的定义中的“恰”字去掉，在转移方程中就要再加入一项f[i][v-1]，这样就可以保证f[N] [V]就是最后的答案。至于为什么这样就可以，由你自己来体会了。

优化空间复杂度

以上方法的时间和空间复杂度均为O(N\*V)，其中时间复杂度基本已经不能再优化了，但空间复杂度却可以优化到O(V)。先考虑上面讲的基本思路如何实现，肯定是有一个主循环i=1..N，每次算出来二维数组f[i][0..V]的所有值。那

么，如果只用一个数组f [0..V]，能不能保证第i次循环结束后f[v]中表示的就是我们定义的状态f[i][v]呢？f[i][v]是由f[i-1][v]和f[i-1] [v-c[i]]两个子问题递推而来，能否保证在推f[i][v]时（也即在第i次主循环中推f[v]时）能够得到f[i-1][v]和f[i-1][v -c[i]]的值呢？事实上，这要求在每次主循环中我们以v=V..0的顺序推f[v]，这样才能保证推f[v]时f[v-c[i]]保存的是状态f[i -1][v-c[i]]的值。伪代码如下：

for i=1..N

for v=V..0

f[v]=max{f[v],f[v-c[i]]+w[i]};

其中的f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程f[i][v]=max{f[i-1][v],f[i- 1][v-c[i]]}，因为现在的f[v-c[i]]就相当于原来的f[i-1][v-c[i]]。如果将v的循环顺序从上面的逆序改成顺序的话，那么则成了f[i][v]由f[i][v-c[i]]推知，与本题意不符，但它却是另一个重要的背包问题P02最简捷的解决方案，故学习只用一维数组解01背包问题是十分必要的。

总结

01背包问题是最基本的背包问题，它包含了背包问题中设计状态、方程的最基本思想，另外，别的类型的背包问题往往也可以转换成01背包问题求解。故一定要仔细体会上面基本思路的得出方法，状态转移方程的意义，以及最后怎样优化的空间复杂度。

P02: 完全背包问题

题目

有N种物品和一个容量为V的背包，每种物品都有无限件可用。第i种物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本思路

这个问题非常类似于01背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取0件、取1件、取2件……等很多种。如果仍然按照解01背包时的思路，令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：f[i][v]=max{f[i-1][v-k\*c[i]]+k\*w[i]|0<=k\*c[i]<= v}。这跟01背包问题一样有O(N\*V)个状态需要求解，但求解每个状态的时间则不是常数了，求解状态f[i][v]的时间是O(v/c[i])，总的复杂度是超过O(VN)的。

将01背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明01背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是试图改进这个复杂度。

一个简单有效的优化

完全背包问题有一个很简单有效的优化，是这样的：若两件物品i、j满足c[i]<=c[j]且w[i]>=w[j]，则将物品j去掉，不用考虑。这个优化的正确性显然：任何情况下都可将价值小费用高得j换成物美价廉的i，得到至少不会更差的方案。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

转化为01背包问题求解

既然01背包问题是最基本的背包问题，那么我们可以考虑把完全背包问题转化为01背包问题来解。最简单的想法是，考虑到第i种物品最多选V/c [i]件，于是可以把第i种物品转化为V/c[i]件费用及价值均不变的物品，然后求解这个01背包问题。这样完全没有改进基本思路的时间复杂度，但这毕竟给了我们将完全背包问题转化为01背包问题的思路：将一种物品拆成多件物品。

更高效的转化方法是：把第i种物品拆成费用为c[i]\*2^k、价值为w[i]\*2^k的若干件物品，其中k满足c[i]\*2^k<V。这是二进制的思想，因为不管最优策略选几件第i种物品，总可以表示成若干个2^k件物品的和。这样把每种物品拆成O(log(V/c[i]))件物品，是一个很大的改进。

但我们有更优的O(VN)的算法。

O(VN)的算法 这个算法使用一维数组，先看伪代码： for i=1..N for v=0..V f[v]=max{f[v],f[v-c[i]]+w[i]};

你会发现，这个伪代码与P01的伪代码只有v的循环次序不同而已。为什么这样一改就可行呢？首先想想为什么P01中要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果f[i-1][v-c[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-c[i]]，所以就可以并且必须采用v= 0..V的顺序循环。这就是这个简单的程序为何成立的道理。

这个算法也可以以另外的思路得出。例如，基本思路中的状态转移方程可以等价地变形成这种形式：f[i][v]=max{f[i-1][v],f[i][v-c[i]]+w[i]}，将这个方程用一维数组实现，便得到了上面的伪代码。

总结

完全背包问题也是一个相当基础的背包问题，它有两个状态转移方程，分别在“基本思路”以及“O(VN)的算法“的小节中给出。希望你能够对这两个状态转移方程都仔细地体会，不仅记住，也要弄明白它们是怎么得出来的，最好能够自己想一种得到这些方程的方法。事实上，对每一道动态规划题目都思考其方程的意义以及如何得来，是加深对动态规划的理解、提高动态规划功力的好方法。

P03: 多重背包问题

题目

有N种物品和一个容量为V的背包。第i种物品最多有n[i]件可用，每件费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

基本算法

这题目和完全背包问题很类似。基本的方程只需将完全背包问题的方程略微一改即可，因为对于第i种物品有n[i]+1种策略：取0件，取1件……取 n[i]件。令f[i][v]表示前i种物品恰放入一个容量为v的背包的最大权值，则：f[i][v]=max{f[i-1][v-k\*c[i]]+ k\*w[i]|0<=k<=n[i]}。复杂度是O(V\*∑n[i])。

转化为01背包问题

另一种好想好写的基本方法是转化为01背包求解：把第i种物品换成n[i]件01背包中的物品，则得到了物品数为∑n[i]的01背包问题，直接求解，复杂度仍然是O(V\*∑n[i])。

但是我们期望将它转化为01背包问题之后能够像完全背包一样降低复杂度。仍然考虑二进制的思想，我们考虑把第i种物品换成若干件物品，使得原问题中第i种物品可取的每种策略——取0..n[i]件——均能等价于取若干件代换以后的物品。另外，取超过n[i]件的策略必不能出现。

方法是：将第i种物品分成若干件物品，其中每件物品有一个系数，这件物品的费用和价值均是原来的费用和价值乘以这个系数。使这些系数分别为 1,2,4,...,2^(k-1),n[i]-2^k+1，且k是满足n[i]-2^k+1>0的最大整数。例如，如果n[i]为13，就将这种物品分成系数分别为1,2,4,6的四件物品。

分成的这几件物品的系数和为n[i]，表明不可能取多于n[i]件的第i种物品。另外这种方法也能保证对于0..n[i]间的每一个整数，均可以用若干个系数的和表示，这个证明可以分0..2^k-1和2^k..n[i]两段来分别讨论得出，并不难，希望你自己思考尝试一下。

这样就将第i种物品分成了O(log n[i])种物品，将原问题转化为了复杂度为O(V\*∑log n[i])的01背包问题，是很大的改进。

O(VN)的算法

多重背包问题同样有O(VN)的算法。这个算法基于基本算法的状态转移方程，但应用单调队列的方法使每个状态的值可以以均摊O(1)的时间求解。由于用单调队列优化的DP已超出了NOIP的范围，故本文不再展开讲解。我最初了解到这个方法是在楼天成的“男人八题”幻灯片上。

小结

这里我们看到了将一个算法的复杂度由O(V\*∑n[i])改进到O(V\*∑log n[i])的过程，还知道了存在应用超出NOIP范围的知识的O(VN)算法。希望你特别注意“拆分物品”的思想和方法，自己证明一下它的正确性，并用尽量简洁的程序来实现。

P04: 混合三种背包问题

问题

如果将P01、P02、P03混合起来。也就是说，有的物品只可以取一次（01背包），有的物品可以取无限次（完全背包），有的物品可以取的次数有一个上限（多重背包）。应该怎么求解呢？

01背包与完全背包的混合

考虑到在P01和P02中最后给出的伪代码只有一处不同，故如果只有两类物品：一类物品只能取一次，另一类物品可以取无限次，那么只需在对每个物品应用转移方程时，根据物品的类别选用顺序或逆序的循环即可，复杂度是O(VN)。伪代码如下：

for i=1..N

if 第i件物品是01背包

for v=V..0

f[v]=max{f[v],f[v-c[i]]+w[i]};

else if 第i件物品是完全背包

for v=0..V

f[v]=max{f[v],f[v-c[i]]+w[i]};

再加上多重背包

如果再加上有的物品最多可以取有限次，那么原则上也可以给出O(VN)的解法：遇到多重背包类型的物品用单调队列解即可。但如果不考虑超过NOIP范围的算法的话，用P03中将每个这类物品分成O(log n[i])个01背包的物品的方法也已经很优了。

小结

有人说，困难的题目都是由简单的题目叠加而来的。这句话是否公理暂且存之不论，但它在本讲中已经得到了充分的体现。本来01背包、完全背包、多重背包都不是什么难题，但将它们简单地组合起来以后就得到了这样一道一定能吓倒不少人的题目。但只要基础扎实，领会三种基本背包问题的思想，就可以做到把困难的题目拆分成简单的题目来解决。

P05: 二维费用的背包问题

问题

二维费用的背包问题是指：对于每件物品，具有两种不同的费用；选择这件物品必须同时付出这两种代价；对于每种代价都有一个可付出的最大值（背包容量）。问怎样选择物品可以得到最大的价值。设这两种代价分别为代价1和代价2，第i件物品所需的两种代价分别为a[i]和b[i]。两种代价可付出的最大值（两种背包容量）分别为V和U。物品的价值为w[i]。

算法

费用加了一维，只需状态也加一维即可。设f[i][v][u]表示前i件物品付出两种代价分别为v和u时可获得的最大价值。状态转移方程就是：f[i][v][u]=max{f[i-1][v][u],f[i-1][v-a[i]][u-b[i]]+w[i]}。如前述方法，可以只使用二维的数组：当每件物品只可以取一次时变量v和u采用顺序的循环，当物品有如完全背包问题时采用逆序的循环。当物品有如多重背包问题时拆分物品。

物品总个数的限制

有时，“二维费用”的条件是以这样一种隐含的方式给出的：最多只能取M件物品。这事实上相当于每件物品多了一种“件数”的费用，每个物品的件数费用均为1，可以付出的最大件数费用为M。换句话说，设f[v][m]表示付出费用v、最多选m件时可得到的最大价值，则根据物品的类型（01、完全、多重）用不同的方法循环更新，最后在f[0..V][0..M]范围内寻找答案。

另外，如果要求“恰取M件物品”，则在f[0..V][M]范围内寻找答案。

小结

事实上，当发现由熟悉的动态规划题目变形得来的题目时，在原来的状态中加一纬以满足新的限制是一种比较通用的方法。希望你能从本讲中初步体会到这种方法。

P06: 分组的背包问题

问题

有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

算法

这个问题变成了每组物品有若干种策略：是选择本组的某一件，还是一件都不选。也就是说设f[k][v]表示前k组物品花费费用v能取得的最大权值，则有f[k][v]=max{f[k-1][v],f[k-1][v-c[i]]+w[i]|物品i属于第k组}。

使用一维数组的伪代码如下：

for 所有的组k

for v=V..0

for 所有的i属于组k

f[v]=max{f[v],f[v-c[i]]+w[i]}

另外，显然可以对每组中的物品应用P02中“一个简单有效的优化”。

小结

分组的背包问题将彼此互斥的若干物品称为一个组，这建立了一个很好的模型。不少背包问题的变形都可以转化为分组的背包问题（例如P07），由分组的背包问题进一步可定义“泛化物品”的概念，十分有利于解题。

P07: 有依赖的背包问题

简化的问题

这种背包问题的物品间存在某种“依赖”的关系。也就是说，i依赖于j，表示若选物品i，则必须选物品j。为了简化起见，我们先设没有某个物品既依赖于别的物品，又被别的物品所依赖；另外，没有某件物品同时依赖多件物品。

算法

这个问题由NOIP2006金明的预算方案一题扩展而来。遵从该题的提法，将不依赖于别的物品的物品称为“主件”，依赖于某主件的物品称为“附件”。由这个问题的简化条件可知所有的物品由若干主件和依赖于每个主件的一个附件集合组成。

按照背包问题的一般思路，仅考虑一个主件和它的附件集合。可是，可用的策略非常多，包括：一个也不选，仅选择主件，选择主件后再选择一个附件，选择主件后再选择两个附件……无法用状态转移方程来表示如此多的策略。（事实上，设有n个附件，则策略有2^n+1个，为指数级。）

考虑到所有这些策略都是互斥的（也就是说，你只能选择一种策略），所以一个主件和它的附件集合实际上对应于P06中的一个物品组，每个选择了主件又选择了若干个附件的策略对应于这个物品组中的一个物品，其费用和价值都是这个策略中的物品的值的和。但仅仅是这一步转化并不能给出一个好的算法，因为物品组中的物品还是像原问题的策略一样多。

再考虑P06中的一句话： 可以对每组中的物品应用P02中“一个简单有效的优化”。这提示我们，对于一个物品组中的物品，所有费用相同的物品只留一个价值最大的，不影响结果。所以，我们可以对主件i的“附件集合”先进行一次01背包，得到费用依次为0..V-c[i]所有这些值时相应的最大价值f'[0..V-c[i]]。那么这个主件及它的附件集合相当于V-c[i]+1个物品的物品组，其中费用为c[i]+k的物品的价值为f'[k]+w[i]。也就是说原来指数级的策略中有很多策略都是冗余的，通过一次01背包后，将主件i转化为 V-c[i]+1个物品的物品组，就可以直接应用P06的算法解决问题了。

更一般的问题

更一般的问题是：依赖关系以图论中“森林”的形式给出（森林即多叉树的集合），也就是说，主件的附件仍然可以具有自己的附件集合，限制只是每个物品最多只依赖于一个物品（只有一个主件）且不出现循环依赖。

解决这个问题仍然可以用将每个主件及其附件集合转化为物品组的方式。唯一不同的是，由于附件可能还有附件，就不能将每个附件都看作一个一般的01 背包中的物品了。若这个附件也有附件集合，则它必定要被先转化为物品组，然后用分组的背包问题解出主件及其附件集合所对应的附件组中各个费用的附件所对应的价值。

事实上，这是一种树形DP，其特点是每个父节点都需要对它的各个儿子的属性进行一次DP以求得自己的相关属性。这已经触及到了“泛化物品”的思想。看完P08后，你会发现这个“依赖关系树”每一个子树都等价于一件泛化物品，求某节点为根的子树对应的泛化物品相当于求其所有儿子的对应的泛化物品之和。

小结

NOIP2006的那道背包问题我做得很失败，写了上百行的代码，却一分未得。后来我通过思考发现通过引入“物品组”和“依赖”的概念可以加深对这题的理解，还可以解决它的推广问题。用物品组的思想考虑那题中极其特殊的依赖关系：物品不能既作主件又作附件，每个主件最多有两个附件，可以发现一个主件和它的两个附件等价于一个由四个物品组成的物品组，这便揭示了问题的某种本质。

我想说：失败不是什么丢人的事情，从失败中全无收获才是。

P08: 泛化物品

定义

考虑这样一种物品，它并没有固定的费用和价值，而是它的价值随着你分配给它的费用而变化。这就是泛化物品的概念。

更严格的定义之。在背包容量为V的背包问题中，泛化物品是一个定义域为0..V中的整数的函数h，当分配给它的费用为v时，能得到的价值就是h(v)。

这个定义有一点点抽象，另一种理解是一个泛化物品就是一个数组h[0..V]，给它费用v，可得到价值h[V]。

一个费用为c价值为w的物品，如果它是01背包中的物品，那么把它看成泛化物品，它就是除了h(c)=w其它函数值都为0的一个函数。如果它是完全背包中的物品，那么它可以看成这样一个函数，仅当v被c整除时有h(v)=v/c\*w，其它函数值均为0。如果它是多重背包中重复次数最多为n的物品，那么它对应的泛化物品的函数有h(v)=v/c\*w仅当v被c整除且v/c<=n，其它情况函数值均为0。

一个物品组可以看作一个泛化物品h。对于一个0..V中的v，若物品组中不存在费用为v的的物品，则h(v)=0，否则h(v)为所有费用为v的物品的最大价值。P07中每个主件及其附件集合等价于一个物品组，自然也可看作一个泛化物品。

泛化物品的和

如果面对两个泛化物品h和l，要用给定的费用从这两个泛化物品中得到最大的价值，怎么求呢？事实上，对于一个给定的费用v，只需枚举将这个费用如何分配给两个泛化物品就可以了。同样的，对于0..V的每一个整数v，可以求得费用v分配到h和l中的最大价值f(v)。也即f(v)=max{h(k) +l(v-k)|0<=k<=v}。可以看到，f也是一个由泛化物品h和l决定的定义域为0..V的函数，也就是说，f是一个由泛化物品h和 l决定的泛化物品。

由此可以定义泛化物品的和：h、l都是泛化物品，若泛化物品f满足f(v)=max{h(k)+l(v-k)|0<=k<=v}，则称f是h与l的和，即f=h+l。这个运算的时间复杂度是O(V^2)。

泛化物品的定义表明：在一个背包问题中，若将两个泛化物品代以它们的和，不影响问题的答案。事实上，对于其中的物品都是泛化物品的背包问题，求它的答案的过程也就是求所有这些泛化物品之和的过程。设此和为s，则答案就是s[0..V]中的最大值。

背包问题的泛化物品

一个背包问题中，可能会给出很多条件，包括每种物品的费用、价值等属性，物品之间的分组、依赖等关系等。但肯定能将问题对应于某个泛化物品。也就是说，给定了所有条件以后，就可以对每个非负整数v求得：若背包容量为v，将物品装入背包可得到的最大价值是多少，这可以认为是定义在非负整数集上的一件泛化物品。这个泛化物品——或者说问题所对应的一个定义域为非负整数的函数——包含了关于问题本身的高度浓缩的信息。一般而言，求得这个泛化物品的一个子域（例如0..V）的值之后，就可以根据这个函数的取值得到背包问题的最终答案。

综上所述，一般而言，求解背包问题，即求解这个问题所对应的一个函数，即该问题的泛化物品。而求解某个泛化物品的一种方法就是将它表示为若干泛化物品的和然后求之。

小结

本讲可以说都是我自己的原创思想。具体来说，是我在学习函数式编程的 Scheme 语言时，用函数编程的眼光审视各类背包问题得出的理论。这一讲真的很抽象，也许在“模型的抽象程度”这一方面已经超出了NOIP的要求，所以暂且看不懂也没关系。相信随着你的OI之路逐渐延伸，有一天你会理解的。

我想说：“思考”是一个OIer最重要的品质。简单的问题，深入思考以后，也能发现更多。

P09: 背包问题问法的变化

以上涉及的各种背包问题都是要求在背包容量（费用）的限制下求可以取到的最大价值，但背包问题还有很多种灵活的问法，在这里值得提一下。但是我认为，只要深入理解了求背包问题最大价值的方法，即使问法变化了，也是不难想出算法的。

例如，求解最多可以放多少件物品或者最多可以装满多少背包的空间。这都可以根据具体问题利用前面的方程求出所有状态的值（f数组）之后得到。

还有，如果要求的是“总价值最小”“总件数最小”，只需简单的将上面的状态转移方程中的max改成min即可。

下面说一些变化更大的问法。

输出方案

一般而言，背包问题是要求一个最优值，如果要求输出这个最优值的方案，可以参照一般动态规划问题输出方案的方法：记录下每个状态的最优值是由状态转移方程的哪一项推出来的，换句话说，记录下它是由哪一个策略推出来的。便可根据这条策略找到上一个状态，从上一个状态接着向前推即可。

还是以01背包为例，方程为f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}。再用一个数组g[i] [v]，设g[i][v]=0表示推出f[i][v]的值时是采用了方程的前一项（也即f[i][v]=f[i-1][v]），g[i][v]表示采用了方程的后一项。注意这两项分别表示了两种策略：未选第i个物品及选了第i个物品。那么输出方案的伪代码可以这样写（设最终状态为f[N][V]）：

i=N

v=V

while(i>0)

if(g[i][v]==0)

print "未选第i项物品"

else if(g[i][v]==1)

print "选了第i项物品"

v=v-c[i]

另外，采用方程的前一项或后一项也可以在输出方案的过程中根据f[i][v]的值实时地求出来，也即不须纪录g数组，将上述代码中的g[i] [v]==0改成f[i][v]==f[i-1][v]，g[i][v]==1改成f[i][v]==f[i-1][v-c[i]]+w[i]也可。

输出字典序最小的最优方案

这里“字典序最小”的意思是1..N号物品的选择方案排列出来以后字典序最小。以输出01背包最小字典序的方案为例。

一般而言，求一个字典序最小的最优方案，只需要在转移时注意策略。首先，子问题的定义要略改一些。我们注意到，如果存在一个选了物品1的最优方案，那么答案一定包含物品1，原问题转化为一个背包容量为v-c[1]，物品为2..N的子问题。反之，如果答案不包含物品1，则转化成背包容量仍为V，物品为2..N的子问题。不管答案怎样，子问题的物品都是以i..N而非前所述的1..i的形式来定义的，所以状态的定义和转移方程都需要改一下。但也许更简易的方法是先把物品逆序排列一下，以下按物品已被逆序排列来叙述。

在这种情况下，可以按照前面经典的状态转移方程来求值，只是输出方案的时候要注意：从N到1输入时，如果f[i][v]==f[i-v]及f[i][v]==f[i-1][f-c[i]]+w[i]同时成立，应该按照后者（即选择了物品i）来输出方案。

求方案总数

对于一个给定了背包容量、物品费用、物品间相互关系（分组、依赖等）的背包问题，除了再给定每个物品的价值后求可得到的最大价值外，还可以得到装满背包或将背包装至某一指定容量的方案总数。

对于这类改变问法的问题，一般只需将状态转移方程中的max改成sum即可。例如若每件物品均是01背包中的物品，转移方程即为f[i][v]=sum{f[i-1][v],f[i-1][v-c[i]]+w[i]}，初始条件f[0][0]=1。

事实上，这样做可行的原因在于状态转移方程已经考察了所有可能的背包组成方案。

最优方案的总数

这里的最优方案是指物品总价值最大的方案。还是以01背包为例。

结合求最大总价值和方案总数两个问题的思路，最优方案的总数可以这样求：f[i][v]意义同前述，g[i][v]表示这个子问题的最优方案的总数，则在求f[i][v]的同时求g[i][v]的伪代码如下：

for i=1..N

for v=0..V

f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}

g[i][v]=0

if(f[i][v]==f[i-1][v])

inc(g[i][v],g[i-1][v]

if(f[i][v]==f[i-1][v-c[i]]+w[i])

inc(g[i][v],g[i-1][v-c[i]])

如果你是第一次看到这样的问题，请仔细体会上面的伪代码。

小结

显然，这里不可能穷尽背包类动态规划问题所有的问法。甚至还存在一类将背包类动态规划问题与其它领域（例如数论、图论）结合起来的问题，在这篇论背包问题的专文中也不会论及。但只要深刻领会前述所有类别的背包问题的思路和状态转移方程，遇到其它的变形问法，只要题目难度还属于NOIP，应该也不难想出算法。

触类旁通、举一反三，应该也是一个OIer应有的品质吧。

最长公共子序列

何谓子序列？例如，序列Z={B，C，D，B}是序列X={A，B，C，B，D，A，B}的子序列，相应的递增下标序列为{2，3，5，7}。

给定2个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列。

给定2个序列X={x1,x2,…,xm}和Y={y1,y2,…,yn}，找出X和Y的最长公共子序列。

解法：

建立一个二维数组，从（0，0）处开始填表，填完为止，输出最后一个数，即是所求结果

关系式为：

if(a[i]==b[j])

f[i][j]=f[i-1][j-1]+1;

else

f[i][j]=max{ f[i-1][j] , f[i][j-1] };

#define N 1010

char a[N],b[N];

int f[N][N];

int LCS(char a[],char b[])

{

int i,j,len1,len2;

len1=(int)strlen(a);

len2=(int)strlen(b);

memset(f,0,sizeof(f));

for(i=1;i<=len1;i++)

for(j=1;j<=len2;j++)

{

if(a[i-1]==b[j-1])

f[i][j]=f[i-1][j-1]+1;

else

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

}

return f[len1][len2];

}

最长递增子序列

子序列是指从给定字符序列中随意地（不一定连续）去掉若干个字符后形成的字符序列。

递增子序列，自然就是要求子序列是递增的。

例：

有这么一个序列，求其的最长上升子序列：

15 45 25 28 12 30

解法：

将原来的序列存在一个数组f[ ]里；

建立一个数组a，初始化都为0，a[i]表示的是以f[i]为最后一个数的上升子序列的长度；

则可以从i=0开始遍历，a[i]应该是所有满足f[i]>f[k]中a[k]+1最大的数，其中k满足0<=k<=i-1;

填完数组之后，求数组的最大值即可

对上述的例子，

a[0]=1;

a[1]=2;(45大于15)；

a[2]=2;(25大于15，但不大于45)；

a[3]=3;(28大于15，大于25，但不大于45，在15和25中，因为25对应的a[2]较大，所以选择25)

…………

以此类推即可。

参考代码：

int n,a[N],dp[N],res[N];

int LIS()

{

res[0]=1;

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

{

int mx=0;

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

{

if(a[i]>a[j] && res[j]>mx)

mx=res[j];

}

res[i]=mx+1;

}

int s=-INF;

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

if(res[i]>s)

s=res[i];

return s;

}

最长递增子序列O(nlogn)算法：

//dp[i]记录长度为i的LIS的最小末尾值

int n,a[N],dp[N];

int LIS()

{

dp[1]=a[1];

int len=1;

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

{

if(a[i] > dp[len])

dp[++len]=a[i];

else

{

int lef=1,rig=len;

while(lef < rig)

{

int mid=(lef+rig)>>1;

if(a[i] > dp[mid]) lef=mid+1;

else rig=mid;

}

dp[lef]=a[i];

}

}

return len;

}

void solve()

{

scanf("%d",&n);

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

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

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

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

// printf("%d ",dp[i]);

// pfn;

}

关于LIS很重要的定理：

Dilworth定理：对于一个偏序集，最少链划分等于最长反链长度。

Dilworth定理的对偶定理：对于一个偏序集，其最少反链划分数等于其最长链的长度。

最长公共递增子序列

给出两个序列a和b，求序列的最长公共递增子序列

例如 a：1，4，2，5，-12 b：-12，1，2，4，最长的公共递增子序列是2（序列为2，4）。

解决方案：

用一个二维数组f[i][j]表示a序列的前i项，b序列的前j项，并且以b[j]结束的LCIS。

则有：

当a[i] != b[j]时，f[i][j]=f[i-1][j];

当a[i] == b[j]时，f[i][j]=max(f[i-1][k])+1,1<=k<=j-1 && b[j]>b[k];

但是这样的效率为O(n^3)。 不难发现，求f[i-1][k]的时候可以通过一个变量temp来解决，当k从1遍历到j的同时，可以不断更新temp的值，如果a[i]>b[j]时，令temp=max(temp,f[i-1][j])；如果a[i]==b[j]时，令f[i][j]=max+1。

#define N 1010

int a[N],b[N];

int f[N][N];

int LCIS(int a[],int b[],int lena,int lenb)

{

int i,j,temp;

memset(f,0,sizeof(f));

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

{

for(j=1,temp=0;j<=lenb;j++)

{

f[i][j]=f[i-1][j];

if(a[i]>b[j] && f[i-1][j]>temp)

temp=f[i-1][j];

if(a[i] == b[j])

f[i][j]=temp+1;

}

}

temp=0;

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

if(f[lena][i] > temp)

temp=f[lena][i];

return temp;

}

f数组也可以改成一维数组，减少空间。f[i]表示原来的f[i][j]，f[i]的值可以沿用上一次循环的值，所以原来代码中的f[i][j]=f[i-1][j]可以省略。

int f[N];

int LCIS(int a[],int b[],int lena,int lenb)

{

int i,j,temp;

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

{

for(j=1,temp=0;j<=lenb;j++)

{

if(a[i]>b[j] && f[j]>temp)

temp=f[j];

if(a[i]==b[j])

f[j]=temp+1;

}

}

temp=0;

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

if(f[i]>temp)

temp=f[i];

return temp;

}

最大子段和

给定由n个整数（包含负整数）组成的序列a1,a2,...,an，求该序列子段和的最大值。

例如有这么一个序列：-2 11 -4 13 -25 21 ，最大子段和为21。

解法：

建立一个f数组，f[i]表示以a[i]为末项的最大子段和，0<=k<=i，如果前i-1项的最大子段和大于0，则加上a[i]；否则f[i]=a[i]。最后从f数组里求得最大值

状态转移方程：

f[i]=f[i-1]+a[i]; f[i-1]>0

f[i]=a[i]; 否则

#define N 1010

#define M INT\_MIN

int f[N],a[N];//f[i]表示以a[i]为末项的最大子段和，0<=k<=i

//如果前i-1项的最大子段和大于0，则f[i]=f[i-1]+a[i]；否则f[i]=a[i]

int LSS\_DP(int n,int a[])

{

int i,res=M;

f[0]=a[0];

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

{

if(f[i-1]>0)

f[i]=f[i-1]+a[i];

else

f[i]=a[i];

if(f[i]>res)

res=f[i];

}

return res;

}

最大子块和

给出一个矩阵，求出其最大的矩阵和。

解法：

可以把最大矩阵和问题看成是二维的最大子段和问题，将二维化为一维求解即可；

n行数，枚举选择其中连续的几行（用两个for就可以完成），将每一列的数都相加，生成一个一维数组，然后求这个一维数组的最大子段和；然后取最大值。

#define N 1010

#define M INT\_MIN

int f[N],a[N],b[N];

int p[N][N];

int LSS\_DP(int n,int a[])

{

//最大子段和

int i,res=M;

f[0]=a[0];

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

{

if(f[i-1]>0)

f[i]=f[i-1]+a[i];

else

f[i]=a[i];

if(f[i]>res)

res=f[i];

}

return res;

}

int LSB\_DP(int m,int n,int \*\*p)

{

//最大子块和

int sum,i,k,j,res;

res=0;

int b[N];

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

{

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

b[k]=0;

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

{

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

b[k]+=p[j][k];

sum=LSS\_DP(n,b);

if(sum>res)

res=sum;

}

}

return res;

}

记忆化搜索

//hdu 1078

给出一个n\*n的矩阵，每个格子上的数代表奶酪数，起始位置在(0, 0)。每次可以水平或者垂直走，最多走k个单位，要求目标位的奶酪数要比当前位的多。每次到一个地方就把当前的奶酪都吃掉，问最多可以吃掉多少奶酪。

从(0,0)开始搜索，能获取的最多奶酪数等于当前奶酪数+下一步能搜到的最多奶酪数。

#define N 110

int n,k,f[N][N],ans[N][N];

int dir[4][2]={{1,0},{-1,0},{0,1},{0,-1}};

int dfs(int x,int y)

{

if(ans[x][y] != -1)

return ans[x][y];

int s=0;

for(int i=0;i<4;i++)

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

{

int tx=x+dir[i][0]\*j;

int ty=y+dir[i][1]\*j;

if(tx>=0 && ty>=0 && tx<n && ty<n && f[tx][ty]>f[x][y])

s=max(s,dfs(tx,ty));

}

ans[x][y]=s+f[x][y];

return ans[x][y];

}

int main()

{

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

{

if(n==-1 && k==-1)

break;

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

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

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

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

printf("%d\n",dfs(0,0));

}

return 0;

}

//hdu 1978

控制一个机器人从一个棋盘的起始点(1,1)走到棋盘的终点(n,m)，机器人一开始在棋盘的起始点并有起始点所标有的能量，只能往下或者右走，走一次消耗一单位能量，不能停留原地，当选择了一条可行路径后，当他走到这条路径的终点时，他将只有终点所标记的能量。（机器人不一定要走到没能量了才停止，可以在中途停下换路径）问有多少种情况。

#define N 110

int n,m,f[N][N],ans[N][N];

int dfs(int x,int y)

{

if(x>n || x<0 || y>m || y<0)

return 0;

if(x==n-1 && y==m-1)

return 1;

if(ans[x][y] != -1)

return ans[x][y];

int s=0;

for(int i=0;i<=f[x][y];i++)

for(int j=0;j+i <= f[x][y];j++)

{

if(i==0 && j==0)

continue;

s=(s+dfs(x+i,y+j))%10000;

}

ans[x][y]=s;

return s;

}

int main()

{

int t;

scanf("%d",&t);

while(t--)

{

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

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

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

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

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

printf("%d\n",dfs(0,0));

}

return 0;

}

期望类dp

问题描述：

给出这么一个图：有4个点，1和2，2和4，1和3，3和4有边。

有个人在1节点，每一分钟他会沿着边随机走到一个节点或者在原地停留，问他走到4号节点需要平均几分钟？

用Ei(i=1,2,3,4) 表示从i号节点走到4号节点的数学期望值。根据题意对1号节点有

E1=（1/3）\*E1+（1/3）\*E2+（1/3）\*E3+1 ① 表示 他下一分钟可以走到2或者3或在原地1，每个可能概率是1/3 ,注意是下一分钟，故要加上1.

同理我们对节点2，3同样可以列出：

E2=(1/3)\*E1+(1/3)\*E2+(1/3)\*E4+1 ②

E3=(1/3)\*E1+(1/3)\*E3+(1/3)\*E4+1 ③

那E4等于多少呢？ 很明显E4=0 ④，因为他就是要到点4。

这样上面1234式其实就是组成了一组方程组，解方程组就可得出E1！！用高斯消元，复杂度是O(n^3)

从上述例子，我们可总结出如何解决期望类问题：

根据题意，表示出各个状态的期望（上例的Ei，1234）,根据概率公式，列出期望之间的方程，解方程即可。

例：

hdu 4405

题意：飞行棋，掷一次骰子，往前进骰子点数，如果有飞机直接跳到飞机所指位置，当位置大于等于n时表示结束。

题解：

可以根据题意列几个方程，当没有飞机的时候，比如n=3，所列方程为：

dp[0]=1/6\*dp[1]+1/6\*dp[2]+1/6\*dp[3]+1

dp[1]=1/6\*dp[2]+1/6\*dp[3]+1

dp[2]=1/6\*dp[3]+1

dp[3]=0

于是可以总结：如果m=0（没有飞机），则dp[i]=sigma(dp[i+k]/6)+1;(1<=k<=6)，否则dp[i]=dp[f[m]]。

#define N 100010

int n,m,f[N];

double dp[N];

int main()

{

int a,b;

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

{

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

while(m--)

{

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

f[a]=b;

}

dp[n]=0;

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

{

if(f[i] != -1)

dp[i]=dp[f[i]];

else

{

double s=0;

for(int j=1;j<=6 && i+j <= n;j++)

s+=dp[i+j];

dp[i]=s/6.0+1;

}

}

printf("%.4lf\n",dp[0]);

}

return 0;

}

状态压缩dp

例1(hdu4529)：

//在8\*8的八皇后棋盘上，问最多放n个骑士的方案数，任意两个骑士不受攻击，骑士“日”字走法。

//因为骑士走的是“日”字格，所以关联到前两行的状态，这道题棋子个数还有限制，所以可以用f[i][j][p][q]来建立模型，表示第 i 行用了

j 个棋子，第 i 行是 p 状态，第 i - 1 行是 q 状态的最大方案数。

//所以复杂度为8\*10\*(1<<7)\*(1<<7)\*(1<<7)，如果不剪枝肯定超时，仔细分析下有很多种情况都是不可行的。

//用vis[i][p][q]标记第i行p状态，第i-1行q状态是否可行。

char c[10][10];

int n,f[10][12][130][130]; //f[i][j][p][q] 第i行，j个棋子数，i行p状态，i-1行q状态

int huang[10],qinum[130],s[3][10]; //huang记录每行皇后的位置

int vis[10][130][130]; //vis[i][p][q], 第i行p状态，第i-1行q状态下，是否可行

int Getqizi(int a) //统计a状态有多少个1

{

int tmp=0;

while(a)

{

if(a&1)

tmp++;

a>>=1;

}

return tmp;

}

void init() //每种状态多少个1，打表处理

{

for(int i=0;i<128;i++)

qinum[i]=Getqizi(i);

}

void deal(int c,int p,int t) //第c行的p状态映射到s[t]上

{

MEM(s[t]);

int tt=0;

while(p)

{

if(tt == huang[c]) //皇后的位置跳过

s[t][tt++]=-1;

s[t][tt++]=p%2;

p>>=1;

}

}

int jud1(int t) //判断相邻两行是否符合条件

{

for(int i=0;i<8;i++)

if(s[t][i] == 1)

{

if(i-2 >= 0 && s[t+1][i-2] == 1)

return 0;

if(i+2 < 8 && s[t+1][i+2] == 1)

return 0;

}

return 1;

}

int jud2() //判断隔一行是否符合条件

{

for(int i=0;i<8;i++)

if(s[0][i] == 1)

{

if(i-1 >= 0 && s[2][i-1] == 1)

return 0;

if(i+1 < 8 && s[2][i+1] == 1)

return 0;

}

return 1;

}

void solve()

{

MEM(f);memset(vis,-1,sizeof(vis));

for(int i=0;i<128;i++) //预处理第一行

{

f[0][qinum[i]][i][0]++;

}

for(int i=1;i<8;i++) //行数

{

for(int j=0;j<=n;j++) //棋子数

{

for(int p=0;p<128;p++) //枚举第i行的状态

{

int num1=qinum[p];

if(num1 > j) //如果当前棋子数超出范围则不用继续，下面也一样

continue;

deal(i,p,0);

for(int q=0;q<128;q++) //枚举第i-1行的状态

{

int num2=qinum[q];

if(num1+num2 > n)

continue;

deal(i-1,q,1);

if(vis[i][p][q] == -1)

vis[i][p][q]=jud1(0);

if(vis[i][p][q] == 0)

continue;

if(i == 1) //如果是第1行，则不用执行下面的判断

{

f[i][j][p][q]+=f[i-1][j-num1][q][0];

continue;

}

for(int k=0;k<128;k++) //枚举第i-2行状态

{

int num3=qinum[k];

if(num1+num2+num3 > n)

continue;

deal(i-2,k,2);

if(vis[i-1][q][k] == 0 || !jud2())

continue;

f[i][j][p][q]+=f[i-1][j-num1][q][k];

}

}

}

}

}

LL ans=0;

for(int i=0;i<128;i++)

for(int j=0;j<128;j++)

ans+=f[7][n][i][j];

pf64(ans);

}

int main()

{

int T;

sf(T);

init();

while(T--)

{

sf(n);

fr(8) sfs(c[i]);

//记录每行皇后位置

for(int i=0;i<8;i++)

{

for(int j=0;j<8;j++)

{

if(c[i][j] == '\*')

{

huang[i]=j;

break;

}

}

}

solve();

}

return 0;

}

例2(hdu2167)：

//给出n\*n的矩阵，每个单元格上一个数，现从矩阵里取数，使得相加和最大，如果一个数被取了，则周围8个方向的数都不能取。3<=n<=15。

//典型的状态压缩dp，首先筛选出每一行所有可行状态，最多状态数为1579，当n等于15时。

//因为只和上一层状态相关，所以只需要记录当前行的状态，判断相邻两行是否符合条件，巧用位运算来处理。

int n,c[16][16];

int f[16][1600]; //f[i][j]：第i行前j种状态的最优解

int num[16][1600]; //num[i][j]：第i行第j种状态下可取数之和

int pnum[16],pz[16][1600];

//pnum[i]：i\*i的矩阵下的可行状态数，最多有1597种状态，当i等于15时候

//pz[i][j]：i\*i的矩阵下的第j种可行状态

int jud(int a,int tmp) //判断a状态是否可行

{

if(a == 0)

return 1;

int p=a%2;

if(p == 1 && tmp == 1)

return 0;

return jud(a/2,p);

}

void init1() //打表所有矩阵(n从3到15)，每一行的所有可行状态

{

for(int t=3;t<=15;t++)

{

int tot=1<<t;

pnum[t]=0;

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

{

if(!jud(i,0))

continue;

pz[t][pnum[t]++]=i;

}

}

}

int deal(int h,int a) //计算第h行，a状态下可取数之和

{

int t=0,r=0;

while(a)

{

if(a&1)

r+=c[h][t];

a>>=1;t++;

}

return r;

}

void init2() //打表每一行下，每种状态的可取数之和

{

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

for(int j=0;j<pnum[n];j++)

num[i][j]=deal(i,pz[n][j]);

}

int jud2(int j,int k) //判断相邻两行，两种状态是否有冲突(巧用位运算)

{

if(j & k || (j>>1) & k || (j<<1) & k)

return 0;

return 1;

}

void solve()

{

MEM(f);MEM(num);

init2();

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

{

for(int j=0;j<pnum[n];j++) //当前状态

{

if(i == 0) //如果是第一行，则不用看上一行

{

f[0][j]=num[0][j];

continue;

}

for(int k=0;k<pnum[n];k++) //前一行状态

{

if(!jud2(pz[n][j],pz[n][k]))

continue;

f[i][j]=max(f[i][j],f[i-1][k]+num[i][j]);

}

}

}

int ans=0;

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

ans=max(ans,f[n-1][i]);

pf(ans);

}

int main()

{

init1();

char s;

while(sf(c[0][0])!=EOF)

{

n=1;

s=getchar();

while(s != '\n')

{

sf(c[0][n++]);

s=getchar();

}

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

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

sf(c[i][j]);

solve();

}

return 0;

}

数位dp

int num[35],f[35][3];

int dfs(int pos,int s,int e) //pos表示当前处理的位置，s表示当前状态，e表示是否有上界

{

if(pos == -1)

return s == final\_status; //final\_status表示最终状态

if(!e && f[pos][s]!=-1) //如果没有限制，且已经搜过则返回

return f[pos][s];

int res=0;

int u=e?num[pos]:9; //取当前最大能取的数

for(int d=0;d<=u;d++)

{

int t=s;

/\*

\* 更新状态t，状态需要自己定义，状态0表示什么，状态1表示什么......

\*/

res+=dfs(pos-1,t,e && d==u);

}

if(!e)

f[pos][s]=res;

return res;

}

//求出0到a之间符合条件的个数

int solve(int a)

{

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

int len=0;

while(a)

{

num[len++]=a%10;

a/=10;

}

return dfs(len-1,0,1);

}

//sample: 求区间[1, a]内有多少个数不含4

int num[35],f[35][2];

int dfs(int pos,int s,int e){

if(pos == -1)

return s == 0;

if(!e && f[pos][s]!=-1)

return f[pos][s];

int res=0;

int u=e?num[pos]:9;

for(int d=0;d<=u;d++)

{

int t=s;

if(d == 4) t=1;

res+=dfs(pos-1,t,e && d==u);

}

if(!e)

f[pos][s]=res;

return res;

}

int FaultyOdometer(int a){

int len=0;

while(a) {

num[len++]=a%10;

a/=10;

}

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

return dfs(len-1,0,1) - 1;

}

//hdu 2089

//给定区间[n,m]，求有多少个数不含4和62

int num[35];

LL f[35][3];

int dfs(int pos,int s,bool e)

{

if(pos == -1)

return s != 2;

if(!e && f[pos][s]!=-1)

return f[pos][s];

int res=0;

int u=e?num[pos]:9;

for(int d=0;d<=u;d++)

{

int t=s;

if(s == 0 && d == 6)

t=1;

if(s == 1 && d == 2)

t=2;

if(s == 1 && d!=6 && d!=2)

t=0;

if(s == 0 && d == 4)

t=2;

if(s == 1 && d == 4)

t=2;

res+=dfs(pos-1,t,e && d==u);

}

if(!e)

f[pos][s]=res;

return res;

}

int solve(int a)

{

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

int len=0;

while(a)

{

num[len++]=a%10;

a/=10;

}

return dfs(len-1,0,1);

}

int main()

{

int n,m;

while(sf2(n,m)!=EOF && (n||m))

pf(solve(m)-solve(n-1));

return 0;

}

### 十二、搜索&查找

bfs

//hdu1241给出一个二维矩阵，看有多少个'@'的块

#include<iostream>

#include<queue>

using namespace std;

#define N 10050

int dir[8][2] = { { 1, 0 }, { 0, 1 }, { -1, 0 }, { 0, -1 }, { 1, 1 }, { -1, 1 },{ 1, -1 }, { -1, -1 } };

struct Point{

int x,y;

};

queue<Point> q;

Point point, point1, point2;

char c[105][105];

bool vis[105][105];

void bfs(int x1, int y1)

{

int i;

point.x = x1;

point.y = y1;

vis[x1][y1] = true;

q.push(point);

while (!q.empty())

{

point1 = q.front();

q.pop();

for (i = 0; i < 8; i++)

{

point2.x = point1.x + dir[i][0];

point2.y = point1.y + dir[i][1];

if (point2.x >= 0 && point2.y >= 0 && c[point2.x][point2.y] == '@'&& !vis[point2.x][point2.y])

{

q.push(point2);

vis[point2.x][point2.y] = true;

}

}

}

}

int main()

{

int m, n, i, j, temp;

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

{

temp = 0;

memset(vis, false, sizeof(vis));

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

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

scanf("%s", c[i]);

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

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

if (c[i][j] == '@' && !vis[i][j])

{

bfs(i, j);

temp++;

}

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

}

return 0;

}

dfs

//dfs打数组f的全排列，ans记录个数

int f[12]={1,2,3,4,5};

int ans,a[12],vis[12];

void dfs(int tmp)

{

if(tmp == 5)

{

ans++;

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

printf("%d ",a[i]);

printf("\n");

return ;

}

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

{

if(!vis[i])

{

vis[i]=1;

a[tmp]=f[i];

dfs(tmp+1);

vis[i]=0;

}

}

}

void solve()

{

MEM(vis);ans=0;

dfs(0);

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

}

hash

//hdu1043 八数码经典问题

//解法：bfs打表+哈希，从最后状态(13245678x)搜，状态用哈希记录，并记录路径（ps：八数码有很多经典解法，比如\*A,迭代加深,双向bfs，但都用到哈希）

struct Node{

int x,y,ha; //x和y记录‘x’的位置，ha记录哈希值

char m[3][3];

};

int dir[4][2]={{1,0},{-1,0},{0,1},{0,-1}};

int fac[12]={1,1,2,6,24,120,720,5040,40320,362880};

char c[5]="udlr";

char path[N]; //path记录路径

Node t,tmp;

int vis[N],pre[N]; //pre记录前一中状态的哈希值

void fun(Node &t,char s[]) //将s转成t中3\*3的格子

{

int len,i,j;

len=(int)strlen(s);

i=0;j=0;

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

{

if(s[k] < '9' && s[k] >= '0')

t.m[i][j++]=s[k];

else if(s[k] == 'x')

{

t.x=i;

t.y=j;

t.m[i][j++]='x';

}

if(j == 3)

{

i++;

j=0;

}

}

}

int Hash(Node p) //计算p中格子的哈希值

{

char s[12];

int i,j,n,c,k=0;

for(i=0;i<3;i++)

for(j=0;j<3;j++)

s[k++]=p.m[i][j];

s[k]='\0';

for(i=0,n=0;i<8;i++)

{

c=0;

for(j=i+1;j<9;j++)

if(s[j] < s[i])

c++;

n+=c\*fac[8-i];

}

return n;

}

void bfs() //逆序搜索

{

int hash1,hash2;

char s[12]="12345678x";

fun(t,s);

MEM(vis);

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

hash1=Hash(t);

vis[hash1]=1;

t.ha=hash1;

queue<Node> Q;

Q.push(t);

while(!Q.empty())

{

t=Q.front();

Q.pop();

hash1=t.ha;

for(int i=0;i<4;i++)

{

tmp=t;

tmp.x=t.x+dir[i][0];

tmp.y=t.y+dir[i][1];

if(tmp.x < 3 && tmp.x >= 0 && tmp.y < 3 && tmp.y >= 0)

{

char tt=tmp.m[t.x][t.y];

tmp.m[t.x][t.y]=tmp.m[tmp.x][tmp.y];

tmp.m[tmp.x][tmp.y]=tt;

hash2=Hash(tmp);

if(!vis[hash2])

{

vis[hash2]=1;

pre[hash2]=hash1;

path[hash2]=c[i];

tmp.ha=hash2;

Q.push(tmp);

}

}

}

}

}

int main()

{

int h;

char str[51];

bfs();

while(gets(str))

{

fun(tmp,str);

h=Hash(tmp);

if(!vis[h])

{

printf("unsolvable\n");

continue;

}

while(path[h] != -1)

{

printf("%c",path[h]);

h=pre[h];

}

printf("\n");

}

return 0;

}

常见字符串hash算法

// RS Hash Function

unsigned int RSHash(char\* str)

{

unsigned int b = 378551 ;

unsigned int a = 63689 ;

unsigned int hash = 0 ;

while (\*str)

{

hash = hash \* a + (\*str ++ );

a \*= b;

}

return (hash & 0x7FFFFFFF );

}

// JS Hash Function

unsigned int JSHash(char\* str)

{

unsigned int hash = 1315423911 ;

while (\*str)

{

hash ^= ((hash << 5 ) + (\*str ++ ) + (hash >> 2 ));

}

return (hash & 0x7FFFFFFF );

}

// P. J. Weinberger Hash Function

unsigned int PJWHash(char\* str)

{

unsigned int BitsInUnignedInt = (unsigned int )( sizeof (unsigned int)\*8 );

unsigned int ThreeQuarters = (unsigned int )((BitsInUnignedInt\*3 ) / 4 );

unsigned int OneEighth = (unsigned int )(BitsInUnignedInt / 8 );

unsigned int HighBits = (unsigned int )( 0xFFFFFFFF ) << (BitsInUnignedInt - OneEighth);

unsigned int hash = 0 ;

unsigned int test = 0 ;

while (\*str)

{

hash = (hash << OneEighth) + (\*str ++ );

if ((test = hash & HighBits) != 0 ) {

hash = ((hash ^ (test >> ThreeQuarters)) & ( ~ HighBits));

}

}

return (hash & 0x7FFFFFFF );

}

// ELF Hash Function

unsigned int ELFHash(char\* str)

{

unsigned int hash = 0 ;

unsigned int x = 0 ; while (\*str)

{

hash = (hash << 4 ) + (\*str ++ );

if ((x = hash & 0xF0000000L ) != 0 ) {

hash ^= (x >> 24 );

hash &= ~ x;

}

}

return (hash & 0x7FFFFFFF );

}

// BKDR Hash Function

unsigned int BKDRHash(char\* str)

{

unsigned int seed = 131 ; // 31 131 1313 13131 131313 etc..

unsigned int hash = 0 ;

while (\*str)

{

hash = hash\*seed + (\*str ++ );

}

return (hash & 0x7FFFFFFF );

}

// SDBM Hash Function

unsigned int SDBMHash(char\* str)

{

unsigned int hash = 0 ;

while (\*str)

{

hash = (\*str ++ ) + (hash << 6 ) + (hash << 16 ) - hash;

}

return (hash & 0x7FFFFFFF );

}

// DJB Hash Function

unsigned int DJBHash(char\* str)

{

unsigned int hash = 5381 ;

while (\*str)

{

hash += (hash << 5 ) + (\*str ++ );

}

return (hash & 0x7FFFFFFF );

}

// AP Hash Function

unsigned int APHash(char\* str)

{

unsigned int hash = 0 ;

int i;

for (i = 0 ;\*str; i ++ )

{

if ((i & 1 ) == 0 ) {

hash ^= ((hash << 7 ) ^ (\*str ++ ) ^ (hash >> 3 ));

} else {

hash ^= ( ~ ((hash << 11 ) ^ (\*str ++ ) ^ (hash >> 5 )));

}

}

return (hash & 0x7FFFFFFF );

}

二分查找

例1：

//在[lef,rig)区间内查找值v,返回下标，a数组已经按从小到大排序，失败返回-1

int bin\_search(int a[],int lef,int rig,int v)

{

int mid;

while(lef < rig)

{

mid=(lef+rig)>>1;

if(a[mid] == v)

return mid;

if(a[mid] < v)

lef=mid+1;

else

rig=mid;

}

return -1;

}

例2：

//查找大于等于v的第一个值，要求lef<=rig，返回值lef总是合理的

int bin\_search(int a[],int lef,int rig,int v)

{

int mid;

while(lef < rig)

{

mid=(lef+rig)>>1;

if(a[mid] < v)

lef=mid+1;

else

rig=mid;

}

return lef;

}

例3:

//hdu2289

已知圆台的上下圆半径（上圆大下圆小），圆台高度以及圆台内水的体积，求水的高度。

二分结果，已知水的高度，可以按照比例计算出水的上圆半径，然后利用公式计算圆台体积。

圆台体积公式：V=PI\*h\*(r\*r+R\*R+R\*r)/3;

double r,R,h,v,tmp;

double fun(double h1)

{

double r2=(R-r)\*h1/h+r;

return PI\*h1\*(r\*r+r2\*r2+r2\*r)/3;

}

void solve()

{

double mid,res,lef=0,rig=h;

while(rig-lef > eps)

{

mid=(lef+rig)/2;

res=fun(mid);

if(res > v)

rig=mid-eps;

else

lef=mid+eps;

}

printf("%.6lf\n",mid);

}

int main()

{

int T;

scanf("%d",&T);

while(T--)

{

scanf("%lf%lf%lf%lf",&r,&R,&h,&v);

solve();

}

return 0;

}

三分查找

三分适用于求凸性函数

double Calc(double a)

{

//根据题意计算

}

void solve(double sta,double end)

{

double Left, Right;

double mid, midmid;

double mid\_area, midmid\_area;

Left = sta; Right = end;

while (Left + eps < Right)

{

mid = (Left + Right) / 2.0;

midmid = (mid + Right) / 2.0;

mid\_area = Calc(mid);

midmid\_area = Calc(midmid);

if (mid\_area+eps < midmid\_area) Right = midmid; //小于求极小值，大于求极大值

else Left = mid;

}

printf("%.2lf %.2lf\n",mid,mid\_area);

}

### 十三、排序

快速排序

//数组a前n个元素排序:quicksort(0,n,a)

void quicksort(int lef,int rig,int a[])

{

if(rig < lef+2)

return ;

int e=rig,p=lef;

while(lef < rig)

{

while(++lef < e && a[lef] <= a[p]); //'<='升序，改为>=降序

while(--rig > p && a[rig] >= a[p]); //'>='升序，改为<=降序(与上面一起改)

if(lef < rig)

swap(a[lef],a[rig]);

}

swap(a[rig],a[p]);

quicksort(p,rig,a);

quicksort(lef,e,a);

}

归并排序

//数组a前n个元素排序:Mergesort(0,n,a)

void Mergesort(int lef,int rig,int a[])

{

int c[12];

int mid,i,j,tmp,cnt=0;

if(rig > lef+1)

{

mid=(lef+rig)/2;

Mergesort(lef,mid,a);

Mergesort(mid,rig,a);

tmp=lef;

for(i=lef,j=mid;i<mid && j<rig;)

{

if(a[i] < a[j]) //'>'升序，'<'降序

{

c[tmp++]=a[j++];

cnt+=mid-i;

}

else

c[tmp++]=a[i++];

}

if(j < rig)

for(;j<rig;j++)

c[tmp++]=a[j];

else

for(;i<mid;i++)

c[tmp++]=a[i];

for(i=lef;i<rig;i++)

a[i]=c[i];

}

}

希尔排序

//数组a前n个元素排序:shellsort(n,a)

void shellsort(int n,int a[])

{

int i,j,k;

for(i=n;i>=1;i--)

a[i]=a[i-1];

k=n/2;

while(k>=1)

{

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

{

a[0]=a[i];

j=i-k;

while(a[j] > a[0] && j>=0) //'>'升序，'<'降序

{

a[j+k]=a[j];

j=j-k;

}

a[j+k]=a[0];

}

k=k/2;

}

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

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

}

冒泡排序

//数组a前n个元素排序:bubblesort(n,a)

void bubblesort(int n,int a[])

{

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

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

if(a[j] < a[j+1]) ///'>'升序，'<'降序

swap(a[j],a[j+1]);

}

堆排序

//数组a前n个元素排序:heapsort(n,a)

void heap(int a[],int i,int m)//调整堆的函数

{

int j,x=a[i];

j=2\*i;

while(j<=m)

{

if(j<m)

if(a[j] < a[j+1]) //'<'升序，'>'降序

j++;

if(a[j] > x) //'>'升序，'<'降序(与上面一起改)

{

a[i]=a[j];

i=j;

j=2\*i;

}

else

j=m+1;

}

a[i]=x;

}

void heapsort(int n,int a[])

{

int i,v;

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

a[i]=a[i-1];

for(i=n/2;i>=1;i--)

heap(a,i,n);

for(v=n;v>=2;v--)

{

swap(a[1],a[v]);

heap(a,1,v-1);

}

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

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

}

选择排序

//数组a前n个元素排序:choosesort(n,a)

void choosesort(int n,int a[])

{

int i,j,k;

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

{

k=i;

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

if(a[j] > a[k]) //'<'升序，'>'降序

k=j;

swap(a[i],a[k]);

}

}

### 十四、计算几何

计算几何完全模板

int doublecmp(double x){//判断double等于0

if(fabs(x)<eps)return 0;else return x<0?-1:1;

}

struct Point{

double x,y;

Point(double x=0,double y=0):x(x),y(y){}

};

typedef Point Vector;

Vector operator+(Vector a,Vector b){return Vector(a.x+b.x,a.y+b.y);}//向量+向量=向量

Vector operator-(Point a,Point b){return Vector(a.x-b.x,a.y-b.y);}//点-点=向量

Vector operator\*(Vector a,double p){return Vector(a.x\*p,a.y\*p);}//向量\*实数=向量

Vector operator/(Vector a,double p){return Vector(a.x/p,a.y/p);}//向量/实数=向量

bool operator<(const Point&a,const Point&b){return a.x<b.x||(a.x==b.x&&a.y<b.y);}

bool operator==(const Point&a,const Point&b){

return doublecmp(a.x-b.x)==0&&doublecmp(a.y-b.y)==0;

}

bool operator!=(const Point&a,const Point&b){return a==b?false:true;}

struct Segment{

Point a,b;

Segment(){}

Segment(Point \_a,Point \_b){a=\_a,b=\_b;}

bool friend operator<(const Segment& p,const Segment& q){return p.a<q.a||(p.a==q.a&&p.b<q.b);}

bool friend operator==(const Segment& p,const Segment& q){return (p.a==q.a&&p.b==q.b)||(p.a==q.b&&p.b==q.a);}

};

struct Circle{

Point c;

double r;

Circle(){}

Circle(Point \_c, double \_r):c(\_c),r(\_r) {}

Point point(double a)const{return Point(c.x+cos(a)\*r,c.y+sin(a)\*r);}

bool friend operator<(const Circle& a,const Circle& b){return a.r<b.r;}

};

struct Line{

Point p;

Vector v;

double ang;

Line() {}

Line(const Point &\_p, const Vector &\_v):p(\_p),v(\_v){ang = atan2(v.y, v.x);}

bool operator<(const Line &L)const{return ang < L.ang;}

};

double dis(Point a,Point b){return sqrt((a.x-b.x)\*(a.x-b.x)+(a.y-b.y)\*(a.y-b.y));}//求两点间距离

double Dot(Vector a,Vector b){return a.x\*b.x+a.y\*b.y;}//|a|\*|b|\*cosθ 点积

double Length(Vector a){return sqrt(Dot(a,a));}//|a| 向量长度

double Angle(Vector a,Vector b){return acos(Dot(a,b)/Length(a)/Length(b));}//向量夹角θ

double Cross(Vector a,Vector b){return a.x\*b.y-a.y\*b.x;}//叉积 向量围成的平行四边形的面积

double Area2(Point a,Point b,Point c){return Cross(b-a,c-a);}//同上 参数为三个点

double DegreeToRadius(double deg){return deg/180\*PI;}

double GetRerotateAngle(Vector a,Vector b){//向量a顺时针旋转theta度得到向量b的方向

double tempa=Angle(a,Vector(1,0));

if(a.y<0) tempa=2\*PI-tempa;

double tempb=Angle(b,Vector(1,0));

if(b.y<0) tempb=2\*PI-tempb;

if((tempa-tempb)>0) return tempa-tempb;

else return tempa-tempb+2\*PI;

}

Vector Rotate(Vector a,double rad){//向量旋转rad弧度

return Vector(a.x\*cos(rad)-a.y\*sin(rad),a.x\*sin(rad)+a.y\*cos(rad));

}

Vector Normal(Vector a){//计算单位法线

double L=Length(a);

return Vector(-a.y/L,a.x/L);

}

Point GetLineProjection(Point p,Point a,Point b){//点在直线上的投影

Vector v=b-a;

return a+v\*(Dot(v,p-a)/Dot(v,v));

}

Point GetLineIntersection(Point p,Vector v,Point q,Vector w){//求直线交点 有唯一交点时可用

Vector u=p-q;

double t=Cross(w,u)/Cross(v,w);

return p+v\*t;

}

int ConvexHull(Point\* p,int n,Point\* ch){//计算凸包

sort(p,p+n);

int m=0;

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

while(m>1&&Cross(ch[m-1]-ch[m-2],p[i]-ch[m-2])<=0) m--;

ch[m++]=p[i];

}

int k=m;

for(int i=n-2;i>=0;i--){

while(m>k&&Cross(ch[m-1]-ch[m-2],p[i]-ch[m-2])<=0) m--;

ch[m++]=p[i];

}

if(n>0) m--;

return m;

}

double Heron(double a,double b,double c){//海伦公式

double p=(a+b+c)/2;

return sqrt(p\*(p-a)\*(p-b)\*(p-c));

}

bool SegmentProperIntersection(Point a1,Point a2,Point b1,Point b2){//线段规范相交判定

double c1=Cross(a2-a1,b1-a1),c2=Cross(a2-a1,b2-a1);

double c3=Cross(b2-b1,a1-b1),c4=Cross(b2-b1,a2-b1);

return doublecmp(c1)\*doublecmp(c2)<0&&doublecmp(c3)\*doublecmp(c4)<0;

}

double CutConvex(const int n,Point\* poly, const Point a,const Point b, vector<Point> result[3]){//有向直线a b 切割凸多边形

vector<Point> points;

Point p;

Point p1=a,p2=b;

int cur,pre;

result[0].clear();

result[1].clear();

result[2].clear();

if(n==0) return 0;

double tempcross;

tempcross=Cross(p2-p1,poly[0]-p1);

if(doublecmp(tempcross)==0) pre=cur=2;

else if(tempcross>0) pre=cur=0;

else pre=cur=1;

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

tempcross=Cross(p2-p1,poly[(i+1)%n]-p1);

if(doublecmp(tempcross)==0) cur=2;

else if(tempcross>0) cur=0;

else cur=1;

if(cur==pre){

result[cur].push\_back(poly[(i+1)%n]);

}

else{

p1=poly[i];

p2=poly[(i+1)%n];

p=GetLineIntersection(p1,p2-p1,a,b-a);

points.push\_back(p);

result[pre].push\_back(p);

result[cur].push\_back(p);

result[cur].push\_back(poly[(i+1)%n]);

pre=cur;

}

}

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

if(points.size()<2){

return 0;

}

else{

return Length(points.front()-points.back());

}

}

double DistanceToSegment(Point p,Segment s){//点到线段的距离

if(s.a==s.b) return Length(p-s.a);

Vector v1=s.b-s.a,v2=p-s.a,v3=p-s.b;

if(doublecmp(Dot(v1,v2))<0) return Length(v2);

else if(doublecmp(Dot(v1,v3))>0) return Length(v3);

else return fabs(Cross(v1,v2))/Length(v1);

}

bool isPointOnSegment(Point p,Segment s){//点在线段上

return doublecmp(DistanceToSegment(p,s))==0;

}

int isPointInPolygon(Point p, Point\* poly,int n){//点与多边形的位置关系

int wn=0;

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

Point& p2=poly[(i+1)%n];

if(isPointOnSegment(p,Segment(poly[i],p2))) return -1;//点在边界上

int k=doublecmp(Cross(p2-poly[i],p-poly[i]));

int d1=doublecmp(poly[i].y-p.y);

int d2=doublecmp(p2.y-p.y);

if(k>0&&d1<=0&&d2>0)wn++;

if(k<0&&d2<=0&&d1>0)wn--;

}

if(wn) return 1;//点在内部

else return 0;//点在外部

}

double PolygonArea(vector<Point> p){//多边形有向面积

double area=0;

int n=p.size();

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

area+=Cross(p[i]-p[0],p[i+1]-p[0]);

return area/2;

}

int PSLGtoPolygons(Segment arr[],int n,vector<Point>\* Polygons){//通过n/2个线段组成的PSLG求所有多边形 返回个数

int count=n-1;

bool vis[9999];memset(vis,0,sizeof vis);//先求一次外包围 去掉多余线段

Point star=arr[0].a,pre=arr[0].a,cur=arr[0].b,purpose=arr[0].b;vis[0]=true;

int mark;

while(purpose!=star){

double theta=-INF;

for(int i=0;i<n;i++)if(!vis[i]&&arr[i].a==cur&&arr[i].b!=pre){

if(theta<GetRerotateAngle(pre-cur,arr[i].b-cur)){

theta=GetRerotateAngle(pre-cur,arr[i].b-cur);

purpose=arr[i].b;

mark=i;

}

}

vis[mark]=true;count--;

pre=cur;cur=purpose;

}//接下来卷包裹最大角度求多边形

int polyidx=0;

while(count>0){//线段个数不为0

for(int i=0;i<n;i++)if(!vis[i]){//找一个出发线段

star=arr[i].a,pre=arr[i].a,cur=arr[i].b,purpose=arr[i].b;vis[i]=true;count--;

Polygons[polyidx].clear();

Polygons[polyidx].push\_back(cur);

break;

}

while(purpose!=star){

double theta=-INF;

for(int i=0;i<n;i++)if(!vis[i]&&arr[i].a==cur&&arr[i].b!=pre){

if(theta<GetRerotateAngle(pre-cur,arr[i].b-cur)){

theta=GetRerotateAngle(pre-cur,arr[i].b-cur);

purpose=arr[i].b;

mark=i;

}

}

vis[mark]=true;count--;

Polygons[polyidx].push\_back(purpose);

pre=cur;cur=purpose;

}

polyidx++;

}

return polyidx;

}

int GetLineCircleIntersection(Line L,Circle C,Point& p1,Point& p2){//圆与直线交点 返回交点个数

double a = L.v.x, b = L.p.x - C.c.x, c = L.v.y, d = L.p.y-C.c.y;

double e = a\*a + c\*c, f = 2\*(a\*b+c\*d), g = b\*b + d\*d -C.r\*C.r;

double delta = f\*f - 4\*e\*g;

if(doublecmp(delta) < 0) return 0;//相离

if(doublecmp(delta) == 0) {//相切

p1=p1=C.point(-f/(2\*e));

return 1;

}//相交

p1=(L.p+L.v\*(-f-sqrt(delta))/(2\*e));

p2=(L.p+L.v\*(-f+sqrt(delta))/(2\*e));

return 2;

}

计算几何完全模板（补充）

#define eps 1e-8

#define zero(x) (((x)>0?(x):-(x))<eps)

struct Point{double x,y; int id;};

struct Line{Point a,b;};

//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dot product (P1-P0).(P2-P0)

double dmult(Point p1,Point p2,Point p0){

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

}

double dmult(double x1,double y1,double x2,double y2,double x0,double y0){

return (x1-x0)\*(x2-x0)+(y1-y0)\*(y2-y0);

}

//两点距离

double distance(Point p1,Point 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));

}

//判点是否在线段上,包括端点

int dot\_online\_in(Point p,Line l){

return zero(xmult(p,l.a,l.b))&&(l.a.x-p.x)\*(l.b.x-p.x)<eps&&(l.a.y-p.y)\*(l.b.y-p.y)<eps;

}

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;

}

int dot\_online\_in(double x,double y,double x1,double y1,double x2,double y2){

return zero(xmult(x,y,x1,y1,x2,y2))&&(x1-x)\*(x2-x)<eps&&(y1-y)\*(y2-y)<eps;

}

//判点是否在线段上,不包括端点

int dot\_online\_ex(Point p,Line l){

return dot\_online\_in(p,l)&&(!zero(p.x-l.a.x)||!zero(p.y-l.a.y))&&(!zero(p.x-l.b.x)||!zero(p.y-l.b.y));

}

int dot\_online\_ex(Point p,Point l1,Point l2){

return dot\_online\_in(p,l1,l2)&&(!zero(p.x-l1.x)||!zero(p.y-l1.y))&&(!zero(p.x-l2.x)||!zero(p.y-l2.y));

}

int dot\_online\_ex(double x,double y,double x1,double y1,double x2,double y2){

return dot\_online\_in(x,y,x1,y1,x2,y2)&&(!zero(x-x1)||!zero(y-y1))&&(!zero(x-x2)||!zero(y-y2));

}

//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);

}

//判两线段相交,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;

}

判断两条线段是否相交

struct point{

double x, y;

};

struct line{

point a, b;

};

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

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

}

int lsinterls(line u,line v)

{

return( (Max(u.a.x,u.b.x)>=Min(v.a.x,v.b.x))&&

(Max(v.a.x,v.b.x)>=Min(u.a.x,u.b.x))&&

(Max(u.a.y,u.b.y)>=Min(v.a.y,v.b.y))&&

(Max(v.a.y,v.b.y)>=Min(u.a.y,u.b.y))&&

(xmulti(v.a,u.b,u.a)\*xmulti(u.b,v.b,u.a)>=0)&&

(xmulti(u.a,v.b,v.a)\*xmulti(v.b,u.b,v.a)>=0));

}

求点到矩形的最短距离和最长距离

double dismin,dismax;//记录点到矩形的最短距离和最长距离

double min(double a,double b){return a<b?a:b;}

double max(double a,double b){return a>b?a:b;}

double dis(double x1,double y1,double x2,double y2)

{

return sqrt((x1-x2)\*(x1-x2)+(y1-y2)\*(y1-y2));

}

void fun(double x,double y,double x1,double y1,double x2,double y2)

{

//将对角线坐标统一成左上和右下

double temp;

if(x1 > x2)

{

temp=x1;

x1=x2;

x2=temp;

}

if(y2 > y1)

{

temp=y2;

y2=y1;

y1=temp;

}

//10种情况分别讨论

if(x<x1 && y>y1)

{

dismin=dis(x,y,x1,y1);

dismax=dis(x,y,x2,y2);

return ;

}

if((x<=x2 && x>=x1) && y>=y1)

{

dismin=y-y1;

dismax=max(dis(x,y,x1,y2),dis(x,y,x2,y2));

return ;

}

if(x>x2 && y>y1)

{

dismin=dis(x,y,x2,y1);

dismax=dis(x,y,x1,y2);

return ;

}

if(x>=x2 && (y<=y1 && y>=y2))

{

dismin=x-x2;

dismax=max(dis(x,y,x1,y1),dis(x,y,x1,y2));

return ;

}

if(x>x2 && y<y2)

{

dismin=dis(x,y,x2,y2);

dismax=dis(x,y,x1,y1);

return ;

}

if((x<=x2 && x>=x1) && y<=y2)

{

dismin=y2-y;

dismax=max(dis(x,y,x1,y1),dis(x,y,x2,y1));

return ;

}

if(x<x1 && y<y2)

{

dismin=dis(x,y,x1,y2);

dismax=dis(x,y,x2,y1);

return ;

}

if(x<=x1 && (y<=y1 && y>=y2))

{

dismin=x1-x;

dismax=max(dis(x,y,x2,y1),dis(x,y,x2,y2));

return ;

}

//点在矩形内

dismin=min(min(x-x1,x2-x),min(y1-y,y-y2));

dismax=max(max(dis(x,y,x1,y1),dis(x,y,x2,y1)),max(dis(x,y,x1,y2),dis(x,y,x2,y2)));

}

求覆盖三点的最小圆

/\*

\* 已知三点，求覆盖三点的最小圆

\* 如果三点构成的三角形是钝角三角形，则圆的直径为三条边的最大值；

\* 否则，圆为三角形的外接圆

\*/

struct Point{

double x,y;

}p[5];

double dist(Point p1,Point p2){

return (p1.x-p2.x)\*(p1.x-p2.x)+(p1.y-p2.y)\*(p1.y-p2.y);

}

void solve()

{

//cen求得圆的圆心，r求得圆的半径

Point cen;

double r;

double t1=dist(p[0],p[1]);

double t2=dist(p[1],p[2]);

double t3=dist(p[0],p[2]);

double tmp=Max(t1,t2,t3);

if(tmp >= t1+t2+t3-tmp)//判断是不是钝角三角形

{

if(tmp == t1)

{

cen.x=(p[0].x+p[1].x)/2;

cen.y=(p[0].y+p[1].y)/2;

}

else if(tmp == t2)

{

cen.x=(p[1].x+p[2].x)/2;

cen.y=(p[1].y+p[2].y)/2;

}

else

{

cen.x=(p[0].x+p[2].x)/2;

cen.y=(p[0].y+p[2].y)/2;

}

r=sqrt(tmp)/2;

}

else //利用公式求外接圆的圆心

{

double k1=(p[0].y-p[1].y)/(p[0].x-p[1].x);

double k2=(p[0].y-p[2].y)/(p[0].x-p[2].x);

double u=(p[0].x\*p[0].x-p[1].x\*p[1].x+p[0].y\*p[0].y-p[1].y\*p[1].y)/(2\*p[0].x-2\*p[1].x);

double v=(p[0].x\*p[0].x-p[2].x\*p[2].x+p[0].y\*p[0].y-p[2].y\*p[2].y)/(2\*p[0].x-2\*p[2].x);

cen.x=v-(u-v)\*k2/(k1-k2);

cen.y=(u-v)/(k1-k2);

r=sqrt(p[0],cen);

}

}

求空间直线间距离

struct point{double x,y,z;};

struct line{point a,b;};

double dmult(point u,point v){return u.x\*v.x+u.y\*v.y+u.z\*v.z;} //三维点积

double vlen(point p){return sqrt(p.x\*p.x+p.y\*p.y+p.z\*p.z);} //向量大小

point subt(point u,point v) //矢量差U-V

{

point ret;

ret.x=u.x-v.x;

ret.y=u.y-v.y;

ret.z=u.z-v.z;

return ret;

}

point xmult(point u,point v) //三维叉积

{

point 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;

}

double linetoline(line u,line v) //空间直线间距离

{

point n=xmult(subt(u.a,u.b),subt(v.a,v.b));

return fabs(dmult(subt(u.a,v.a),n))/vlen(n);

}

求两个圆相交面积

struct point{double x,y;};

double dist(point a,point b)

{return sqrt((a.x-b.x)\*(a.x-b.x)+(a.y-b.y)\*(a.y-b.y));}

double cir\_area(point a,double r1,point b,double r2)

{

double d=dist(a,b);

if(r1+r2 <= d || r1==0 || r2==0) //如果相离或者相切，结果为0

return 0;

else if(d <= fabs(r1-r2)) //如果包含，结果为面积小的圆

{

double rr=min(r1,r2);

return PI\*rr\*rr;

}

else

{

double ang1=acos((r1\*r1+d\*d-r2\*r2)/(2\*r1\*d)); //圆1和圆心距形成的夹角

double ang2=acos((r2\*r2+d\*d-r1\*r1)/(2\*r2\*d)); //圆2和圆心距形成的夹角

double shan1=r1\*r1\*ang1; //圆1和公共弦形成的扇形

double shan2=r2\*r2\*ang2; //圆2和公共弦形成的扇形

double are\_tri1=r1\*r1\*sin(ang1)\*cos(ang1); //圆1和公共弦形成的三角形

double are\_tri2=r2\*r2\*sin(ang2)\*cos(ang2); //圆2和公共弦形成的三角形

double ans=shan1+shan2-are\_tri1-are\_tri2; //计算公共面积

return ans;

}

}

求两空间异面直线公垂线及交点坐标

struct Point{

double x,y,z;

};

int tt;

Point aa,bb,cc,dd;

Point diana,dianb;

double fun(double a,double b,double c,double d){

return(a\*d-b\*c);

}

Point jiaodian(Point A,Point B,Point C,Point D)

{

double xa,ya,za,xb,yb,zb,xc,yc,zc,xd,yd,zd,e,f,g,h,i,j,k,l,m,n,o,p,q,kk;

Point ss;

xa=A.x;ya=A.y;za=A.z;

xb=B.x;yb=B.y;zb=B.z;

xc=C.x;yc=C.y;zc=C.z;

xd=D.x;yd=D.y;zd=D.z;

e=fun(yb-ya,zb-za,yd-yc,zd-zc);

f=fun(zb-za,xb-xa,zd-zc,xd-xc);

g=fun(xb-xa,yb-ya,xd-xc,yd-yc);

h=xb-xa;i=yb-ya;j=zb-za;

k=xd-xc;l=yd-yc;m=zd-zc;

n=h\*i\*l-i\*i\*k-j\*j\*k+h\*j\*m;

o=h\*h\*l-h\*i\*k-i\*j\*m+j\*j\*l;

p=h\*j\*k-h\*h\*m-i\*i\*m+i\*j\*l;

q=-xa\*n+ya\*o-za\*p;

kk=(o\*yc-n\*xc-p\*zc-q)/(n\*k-o\*l+p\*m);

ss.x=k\*kk+xc;

ss.y=l\*kk+yc;

ss.z=m\*kk+zc;

return(ss);

}

void Print(Point pp){

printf("%.6lf %.6lf %.6lf",pp.x,pp.y,pp.z);

}

double dist(Point p,Point q)

{ return(sqrt(((p.x-q.x)\*(p.x-q.x))+((p.y-q.y)\*(p.y-q.y))+((p.z-q.z)\*(p.z-q.z))));

}

void solve()

{

scanf("%lf%lf%lf%lf%lf%lf",&aa.x,&aa.y,&aa.z,&bb.x,&bb.y,&bb.z);

scanf("%lf%lf%lf%lf%lf%lf",&cc.x,&cc.y,&cc.z,&dd.x,&dd.y,&dd.z);

diana=jiaodian(cc,dd,aa,bb);

dianb=jiaodian(aa,bb,cc,dd);

printf("%.6lf\n",dist(diana,dianb));

Print(diana);

printf(" ");

Print(dianb);

printf("\n");

}

求三角形整数内点个数

/\*

\* 求三角形内坐标为整数的点个数，不包含边和顶点

\* point存三角形坐标，这里3个点为(0,0),(n,m),(p,0)

\*/

struct Point{

int x,y;

}point[3];

int n,m,p;

int gcd(int a,int b){return b?gcd(b,a%b):a;}

double triangle\_area(Point t[]) {

return fabs(t[0].x\*t[1].y+t[1].x\*t[2].y+t[2].x\*t[0].y-t[1].x\*t[0].y-t[2].x\*t[1].y-t[0].x\*t[2].y)/2;

}

int solve(int n,int m,int p){

int t1=gcd(n,m);

int t2=gcd(abs(n-p),m);

int t3=p;

point[0].x=0;point[0].y=0;

point[1].x=n;point[1].y=m;

point[2].x=p;point[2].y=0;

double area=triangle\_area(point);

return (int)(area-(t1+t2+t3)/2+1);

}

凸包

#define N 100010

struct point{

double x,y;

}p[N],res[N];

int tot,n;

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

return l.y < r.y || (l.y == r.y && l.x < r.x);

}

double dist(point a,point b){

return sqrt((a.x-b.x)\*(a.x-b.x)+(a.y-b.y)\*(a.y-b.y));

}

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

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

}

int graham(point pnt[], int n, point res[])

{

int i, len, top = 1;

sort(pnt, pnt + n);

if (n == 0) return 0; res[0] = pnt[0];

if (n == 1) return 1; res[1] = pnt[1];

if (n == 2) return 2; res[2] = pnt[2];

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

{

while (top && mult(pnt[i], res[top], res[top-1]))

top--;

res[++top] = pnt[i];

}

len = top; res[++top] = pnt[n - 2];

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

{

while (top!=len && mult(pnt[i], res[top],res[top-1]))

top--;

res[++top] = pnt[i];

}

return top;

}

void solve()

{

tot=0;

double a,b;

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

{

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

p[tot].x=a;

p[tot++].y=b;

}

int tmp=graham(p,tot,res);

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

}

已知球面两点经纬度求距离

double deal(double a1,double b1,double a2,double b2,double r) //a为经度 b为纬度，且都是弧度，r是球面半径

{

return r\*acos(cos(b1)\*cos(b2)\*cos(a1-a2)+sin(b1)\*sin(b2));

}

最近点对问题

struct Point{

double x,y;

int index;

};

Point a[N],b[N],c[N];

inline double min(double p,double q){return (p>q)?(q):(p);}

double dis(Point a,Point b){return sqrt((a.x-b.x)\*(a.x-b.x)+(a.y-b.y)\*(a.y-b.y));}

int cmp\_x(const void \*p,const void \*q)//把点按照 x 坐标排序

{

double temp = ((Point \*)p)->x - ((Point \*)q)->x;

if(temp > 0) return 1;

else if(fabs(temp) < eps) return 0;

else return -1;

}

int cmp\_y(const void \*p,const void \*q)//把点按照 y 坐标排序

{

double temp = ((Point \*)p)->y - ((Point \*)q)->y;

if(temp > 0) return 1;

else if(fabs(temp) < eps) return 0;

else return -1;

}

int merge(Point p[],Point q[],int s,int m,int t)

{//把 q[s~m],q[m+1~t]合并回 p , 而且按照 y 排序。q[s~m],q[m+1~t]已经是按照 y 排序的了

int i,j,k;

for(i = s, j = m+1, k = s;i <= m && j <= t;)

{

if(q[i].y > q[j].y) p[k++] = q[j],j++;

else p[k++] = q[i],i++;

}

while( i <= m) p[k++] = q[i++];

while( j <= t) p[k++] = q[j++];

memcpy(q+s,p+s,(t-s+1)\*sizeof(p[0]));//把 p 复制 给 q

return 0;

}

double closest(Point a[],Point b[],Point c[],int p,int q)

{//a是按照 x 坐标从小到大排序的，b 是按照 y 坐标从小到大排序的

//求 a 从 p 点到 q 点的最小点对距离

if(q - p == 1) return dis(a[p],a[q]);//两个点

if(q - p == 2) //三个点的情况

{

double x1 = dis(a[p],a[q]);

double x2 = dis(a[p+1],a[q]);

double x3 = dis(a[p],a[p+1]);

if(x1 < x2 && x1 < x3) return x1;

else if(x2 < x3) return x2;

else return x3;

}

int i,j,k,m = (p+q) / 2;//按照横坐标分成两部分

double d1,d2;

//把 点分成两部分，左边的存在 c[p~m],右边的存在 c[m+1~q],两部分均按照 y 排序

for(i = p,j = p,k = m + 1;i <= q;i++)

if(b[i].index <= m) c[j++] = b[i];

else c[k++] = b[i];

d1 = closest(a,c,b,p,m);//左半部分的最小点对距离

d2 = closest(a,c,b,m+1,q);//右半部分的最小点对距离

double dm = min(d1,d2);//取两者的较小值

//数组 c 左右部分分别是对　ｙ坐标有序的，合并回　ｂ

merge(b,c,p,m,q);

for(i = p,k = p ; i <= q;i++)

if(fabs(b[i].x - b[m].x) < dm) c[k++] = b[i];

//找出离划分基准左右不超过dm的部分，且仍然对 y 有序

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

for(j = i+1;j < k && c[j].y -c[i].y < dm;j++)

{

double temp = dis(c[i],c[j]);

if(temp < dm) dm = temp;

}

return dm;

}

void solve(int n)

{

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

scanf("%lf%lf",&a[i].x,&a[i].y);

qsort(a,n,sizeof(a[0]),cmp\_x);

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

a[i].index=i;

memcpy(b,a,n\*sizeof(a[0]));

qsort(b,n,sizeof(b[0]),cmp\_y);

double d=closest(a,b,c,0,n-1);

printf("%.2lf\n",d);

}