目录

[一、 字符串 2](#_Toc491287325)

[1、KMP 2](#_Toc491287326)

[2、AC自动机 3](#_Toc491287327)

[3、Manacher 5](#_Toc491287328)

[4、Trie 6](#_Toc491287329)

[5、回文树 9](#_Toc491287330)

[6、后缀数组 10](#_Toc491287331)

[二、 图论 14](#_Toc491287332)

[1、 最短路Bellman-Ford 14](#_Toc491287333)

[2、 最短路 Dijkstra 16](#_Toc491287334)

[3、最短路 Floyd 18](#_Toc491287335)

[4、最小生成树 Kruskal 18](#_Toc491287336)

[5、LCA 20](#_Toc491287337)

[6、最大团 23](#_Toc491287338)

[7、最大流（只保留Dinic） 24](#_Toc491287339)

[8、二分图匹配 25](#_Toc491287340)

[三、 数据结构 27](#_Toc491287341)

[1、 BIT 27](#_Toc491287342)

[2、 并查集 28](#_Toc491287343)

[四、 数学 28](#_Toc491287344)

[1、 FFT 28](#_Toc491287345)

[2、 改进版FFT 29](#_Toc491287346)

[3、 NTT 及模任意数 32](#_Toc491287347)

[4、 二次剩余 36](#_Toc491287348)

[5、 逆元及求组合数 38](#_Toc491287349)

[6、 同种不相邻的排列数 39](#_Toc491287350)

[7、 原根 43](#_Toc491287351)

[8、 中国剩余定理 45](#_Toc491287352)

[9、 莫比乌斯反演几例 45](#_Toc491287353)

# 字符串

## 1、KMP

/\*

KMP基本模板

\*/

#include <iostream>

#include <string>

#include <algorithm>

#include <cstring>

#include <cstdio>

#include <cmath>

#include <queue>

#include <set>

#include <map>

#include <list>

#include <stack>

#define mp make\_pair

#define Elemtype int

typedef long long ll;

typedef unsigned long long ull;

const int MAX=1e6+5;

const int INF=-1e9;

using namespace std;

typedef pair<int,int> pii;

int f[MAX];

string x;

void getf(string x,int m)//m为串的长度

{

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

for(int i=2,j=0;i<=m;i++)

{

while(j&&x[j+1]!=x[i])

j=f[j];

if(x[j+1]==x[i])

j++;

f[i]=j;

}

}

bool kmp(string x,string y)//在y中匹配x

{

getf(x,x.size());

for(int i=1,j=0;i<=y.size();i++)

{

while(j&&x[j+1]!=y[i])

j=f[j];

if(x[j+1]==y[i])

j++;

if(j>=x.size())

return true;

}

return false;

}

## 2、AC自动机

const int SIGMA\_SIZE=26;

const int MAXNODE=11000;

const int MAXS=160;

/\*

由于失配过程比较复杂，要反复沿着失配边走，

在实践中常常会把下述自动机改造一下，把所有不存在的边“补上”

即把计算失配函数中的语句if(!u)continue;改成

if(!u){ch[r][c]=ch[f[r]][c];continue;}

这样，就完全不需要失配函数，而是对所有转移一视同仁

即find函数中的while(j&&!ch[j][c])j=f[j] 可以直接删除

\*/

struct AhoCorasickAutomata

{

int ch[MAXNODE][SIGMA\_SIZE];

int f[MAXNODE];//fail函数

int val[MAXNODE];//每个字符串的结尾结点都有非0的val

int last[MAXNODE];//输出链表的下一个单词结点

int num;//trie树编号（亦为包含根结点的当前size）

//初始化

void init()

{

num=1;

memset(ch[0],0,sizeof(ch[0]));

}

//返回字符对应编号

int idx(char c)

{

return c-'a';

}

//插入权值为v的字符串

void insert(char \*s,int v)

{

int u=0,n=strlen(s);

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

{

int c=idx(s[i]);

if(!ch[u][c])

{

memset(ch[num],0,sizeof(ch[num]));

val[num]=0;

ch[u][c]=num++;

}

u=ch[u][c];

}

val[u]=v;

}

//递归打印以结点j结尾的所有字符串

void print(int j)

{

if(j)

{

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

print(last[j]);

}

}

//计算fail函数

void getFail()

{

queue <int> que;

f[0]=0;

for(int c=0;c<SIGMA\_SIZE;c++)

{

int u=ch[0][c];

if(u)

{

f[u]=0;que.push(u);last[u]=0;

}

}

while(!que.empty())

{

int r=que.front();que.pop();

for(int c=0;c<SIGMA\_SIZE;c++)

{

int u=ch[r][c];

if(!u)

continue;

que.push(u);

int v=f[r];

while(v&&!ch[v][c])//类似kmp的过程

v=f[v];

f[u]=ch[v][c];

last[u]=val[f[u]]?f[u]:last[f[u]];

}

}

}

//在T中中模板

int find(char \*T)

{

int n=strlen(T);

int u=0;//当前结点编号,初始为根结点

for(int i=0;i<n;i++)//文本串当前指针

{

int c=idx(T[i]);

while(u&&!ch[u][c])//顺着失配指针走，直到可以匹配或回到根节点

u=f[u];

u=ch[u][c];

if(val[u])

print(u);

else if(last[u])//不是很明白该else的含义

print(last[u]);

}

}

};

## 3、Manacher

/\*

Manacher 最长回文子串

O(len)

\*/

char now[2\*MAX+5];//至少是初始串长度的2倍

int rl[2\*MAX+5];//对于下标i 表示以i为中心的最长回文串半径

int MAXRIGHT,pos;//当前涉及到的回文串中达到最右的位置的下一个位置，及其中心

int Manacher(char \*s)//下标从0开始

{

MAXRIGHT=pos=0;

int MAXLEN=0;//最长回文串长度

int len=strlen(s);

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

now[2\*i]='#',now[2\*i+1]=s[i];// 任意选定串中不出现的字符

now[len=2\*len]='#';

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

{

/\*

i的位置必定在pos右侧 只需分情况讨论与MAXRIGHT的位置关系

\*/

if(i<MAXRIGHT)

rl[i]=min(rl[2\*pos-i],MAXRIGHT-i);

else

rl[i]=1;

while(i-rl[i]>=0&&i+rl[i]<=len&&now[i-rl[i]]==now[i+rl[i]])

++rl[i];

if(i+rl[i]>MAXRIGHT)

MAXRIGHT=i+rl[i],pos=i;

MAXLEN=max(MAXLEN,rl[i]);

}

return MAXLEN-1;

}

## 4、Trie

/\*极其容易MLE版本 \*/

struct Trie

{

int ch[MAX\_NODE][sigma\_size];//点数、“字母”数

int val[MAX\_NODE];

int num;//结点总数

Trie(){num=1;memset(ch[0],0,sizeof(ch[0]));}//初始时仅有根节点

int idx(char c)//返回对应字符的编号

{

return c-'a';

}

/\*

clear函数，初始化trie

\*/

void clear() { num = 1; memset(ch[0], 0, sizeof(ch[0])); }

/\*

插入字符串s,附加信息为v。注意v必须非0，因为0代表：本结点不是单词结点

\*/

void insert(char \*s,int v)

{

int u=0,len=strlen(s);

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

{

int c=idx(s[i]);

if(!ch[u][c])//结点不存在

{

memset(ch[num],0,sizeof(ch[num]));

val[num]=0;//中间节点的附加信息为0

ch[u][c]=num++;//新建结点

}

u=ch[u][c];//往下走

}

val[u]=v;//字符串的最后一个字符的附加信息为v

}

/\*

查询字符串的“附加信息”

查询过程中间中断返回0

\*/

int check(char \*s)

{

int u=0,len=strlen(s);

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

{

int c=idx(s[i]);

if(!ch[u][c])

return 0;

u=ch[u][c];

}

return val[u];

}

/\*

找字符串s的长度不超过len的前缀

\*/

void find\_prefixes(const char \*s,int len,vector <int> &ans)

{

int u=0;

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

{

if(s[i]=='\0')

break;

int c=idx(s[i]);

if(!ch[u][c])

break;

u=ch[u][c];

if(val[u]!=0)//过程中所有找到的全都push进去

ans.push\_back(val[u]);

}

}

};

/\*不易MLE版本 \*/

struct Trie {

bool isWord;

Trie\* child[26];

Trie(bool isWord):isWord(isWord)

{

memset(child,0,sizeof(child));

}

void addWord(string &s)

{

Trie\*cur =this;

for(char c: s)

{

Trie\* next=cur->child[c-'a'];

if(next==nullptr)

next=cur->child[c-'a']=new Trie(false);

cur=next;

}

cur->isWord=true;

}

~Trie()

{

for(int i=0;i<26;++i)

{

if(child[i])

delete child[i];

}

}

};

## 5、回文树

/\*

回文树

空间复杂度 O(N\*字符集大小）

时间复杂度 O（N\*log(字符集大小））

\*/

const int MAXN=1e5+5;

const int N=26;//字符集大小

struct Palindromic\_Tree

{

int next[MAXN][N];

//next指针和字典树类似，指向的串由当前串两端加上同一个字符构成

int fail[MAXN];//fail指针，失配后跳转到fail指针指向的节点

int cnt[MAXN];//节点i表示的本质不同的串的个数（建树时求出的是不完全的，count（）函数之后才全）

int num[MAXN];//num[i]表示以节点i表示的最长回文串的最右端点为回文串结尾的回文串个数

int len[MAXN];//len[i]表示节点i表示的回文串的长度

int S[MAXN];//第i次添加的字符（一开始设S[0]=-1(可以是任意一个串S中不会出现的字符））

int last;//指向上一个字符所在的节点，方便下次add

int n;//字符数组指针

int p;//节点

int newnode(int l)//新建节点

{

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

next[p][i]=0;

cnt[p]=0;

num[p]=0;

len[p]=l;

return p++;

}

void init()//初始化

{

p=0;

newnode(0);

newnode(-1);

last=0;

n=0;

S[n]=-1;//开始放一个字符集中没有的字符，减少特判

fail[0]=1;

}

int get\_fail(int x)//和KMP一样，失配后找一个尽量最长的

{

while(S[n-len[x]-1]!=S[n])

x=fail[x];

return x;

}

void add(int c)

{

c-='a';

s[++n]=c;

int cur=get\_fail(last);//通过上一个回文串找这个回文串匹配位置

if(!next[cur][c])//若该回文串未出现过，说明出现了一个新的本质不同的回文串

{

int now=newnode(len[cur]+2);//新建节点

fail[now]=next[get\_fail(fail[cur])][c];

next[cur][c]=now;

num[now]=num[fail[now]]+1;

}

last=next[cur][c];

cnt[last]++;

}

void count()

{

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

cnt[fail[i]]+=cnt[i];

//父亲累加儿子的cnt，因为若fail[v]=u，则u一定是v的回文子串

}

};

## 6、后缀数组

nlogn版

const int MAX=2505;

const ll INF=1e18+5;

const int B=1024;//桶的大小

const double M=4e18;

using namespace std;

const int MOD=1e9+7;

typedef pair<int,int> pii;

char s[MAX];

int sa[MAX],t[MAX],t2[MAX],c[MAX],n;

int rank[MAX],height[MAX];//rank[i]代表后缀i在sa数组中的下标

//height[i]定义为sa[i-1]和sa[i]的最长公共前缀（LCP）的长度

//sa数组储存第i大的起始位置

/\*

对于两个后缀j和k，不妨设rank[j]<rank[k]，则不难证明后缀j和k的LCP长度等于

height[rank[j]+1],height[rank[j]+2]……height[rank[k]]中的最小值

即RMQ(height,rank[j]+1,rank[k])

\*/

void build\_sa(int m)

{

int i,\*x=t,\*y=t2;//x记录各个下标开始的串的排序编号（大小）

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

c[i]=0;//初始化c

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

++c[x[i]=s[i]];//该字符出现次数++

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

c[i]+=c[i-1];//求前缀和，sa中每个字符出现的下标区间得到确定，总区间为[0,n-1]

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

sa[--c[x[i]]]=i;//该下标（排位）对应的位置

for(int k=1;k<=n;k<<=1)//每组个数/2

{

int p=0;

//直接利用sa数组排序第二关键字

for(int i=n-k;i<n;i++)//因为后面总共不足k个

y[p++]=i;//后面的k个一定排序两两不同，

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

if(sa[i]>=k)

y[p++]=sa[i]-k;//恰好是之前涉及不到的

/\*感觉不这样也行 因为后面根本就没有用到？？\*/

//基数排序第一关键字

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

c[i]=0;

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

++c[x[y[i]]];

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

c[i]+=c[i-1];

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

sa[--c[x[y[i]]]]=y[i];

swap(x,y);

p=1;

x[sa[0]]=0;

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

x[sa[i]]=y[sa[i-1]]==y[sa[i]]&&y[sa[i-1]+k]==y[sa[i]+k]?p-1:p++;

if(p>=n)

break;//以后继续倍增，sa也不会改变，退出

m=p;//下次基数排序的最大值

}

}

int m;//模板长度，简单起见，这里存在全局变量中

int cmp\_suffix(char \*pattern,int p)//判断模板s是否为后缀p的前缀

{

return strncmp(pattern,s+sa[p],m);

}

int find(char \*P)

{

m=strlen(P);

if(cmp\_suffix(P,0)<0)

return -1;

if(cmp\_suffix(P,n-1)>0)

return -1;

int L=0,R=n-1;

while(L<=R)

{

int M=(L+R)/2;

int res=cmp\_suffix(P,M);

if(!res)

return M;

if(res<0)//模式串比第M大的小

R=M-1;

else//模式串比第M大的大

L=M+1;

}

return -1;//找不到

}

void getHeight()

{

int i,j,k=0;

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

rank[sa[i]]=i;

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

{

if(k)

k--;

int j=sa[rank[i]-1];

while(s[i+k]==s[j+k])

++k;

height[rank[i]]=k;

}

}

2个log版

int n,k;

int rank[MAX];

int tmp[MAX];

//比较rank[i],rank[i+k] 和rank[j],rank[j+k]

bool compare\_sa(int i,int j)

{

if(rank[i]!=rank[j])

return rank[i]<rank[j];

else

{

int ri=i+k<=n?rank[i+k]:-1;

int rj=j+k<=n?rank[j+k]:-1;

return ri<rj;

}

}

//计算字符串S的后缀数组

void construct\_sa(string S,int \*sa)

{

n=S.length();

//初始长度为1,rank直接取字符的编码

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

{

sa[i]=i;

rank[i]=i<n?S[i]:-1;

}

//利用对长度为k的排序的结果对长度为2k的排序

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

{

sort(sa,sa+n+1,compare\_sa);

//先在tmp中临时存储新计算的rank，再转存回rank中

tmp[sa[0]]=0;

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

tmp[sa[i]]=tmp[sa[i-1]]+compare\_sa(sa[i-1],sa[i])?1:0;

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

rank[i]=tmp[i];

}

}

/\*

基于后缀数组的字符串匹配

假设已经计算好了字符串S的后缀数组，现在要求字符串T在字符串S中出现的位置

只要通过二分搜索就可以在O(TlogS)的时间完成

\*/

bool contain(string S,int \*sa,string T)

{

int l=0,r=S.length(),mid;

while(r-l>1)

{

mid=(l+r)/2;

//比较S从位置sa[c]开始长度为T的字串与T

if(S.compare(sa[mid],T.length(),T)==0)

l=mid;

else

r=mid;

}

return S.compare(sa[r],T.length(),T)==0;

}

# 图论

## 最短路Bellman-Ford

/\*

单源最短路

Bellman-Ford算法

设从起点s出发到顶点i的最短距离为d[i].则下述等式成立

d[i]=min{d[j]+（从j到i的边的权值）|e=(j,i)∈E}

若满足DAG，可以按拓扑序给顶点编号，利用递推求解

不然，设初值 d[s]=0，d[i]=INF（足够大的常数）

只要无负圈，更新操作就是有限的

\*/

struct edge

{

int from,to,cost;

};

//从顶点from 指向顶点 to的权值为const的边

edge es[MAX\_E];//边

int d[MAX\_V];//最短距离

int V,E;//V是顶点数，E是边数

void shortest\_path(int s)

{

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

d[i]=INF;

d[s]=0;

while(1)

{

bool update=false;

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

{

edge e=es[i];

if(d[e.from]!=INF&&d[e.to]>d[e.from]+e.cost)

{

d[e.to]=d[e.from]+e.cost;

update=true;

}

}

if(!update)

break;

}

}

//求解从顶点s出发到所有点的最短距离

/\*

如果图中不存在从s可到达的负圈，那么最短路不会经过同一个顶点两次

（即最多通过|V|-1条边）

while(1)最多循环|V|-1次

而若有，在第|V|次循环中也会更新d的值，故可用此性质检查负圈

（此时初始时将所有d[i]初始化为0）

\*/

bool find\_negative\_loop()

{

memset(d,0,sizeof(d));

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

{

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

{

edge e=es[j];

if(d[e.to]>d[e.from]+e.cost)

{

d[e.to]=d[e.from]+e.cost;

if(i==V-1)//到第n次仍然更新了，则存在负圈

return true;

}

}

}

return false;

}

//如果返回true则存在负圈

## 最短路 Dijkstra

/\*

单源最短路 2

Dijkstra算法

考虑没有负边的情况

在Bellman-Ford 算法中，如果d[i]还不是最短距离，

即使进行d[j]=d[i]+(从i到j的边的权值)的更新，d[j]也不会变成最短距离

并且，即使d[i]没有变化，每一次循环也要检查一遍从i出发的所有边，这很浪费时间。

故可作如下优化

1、找到最短距离已经确定的顶点，从它出发更新相邻顶点的最短距离

2、此后不需再关心1中的最短距离已经确定的顶点

\*/

int cost[MAX\_V][MAX\_V];//cost[u][v]表示边e=(u,v)的权值（不存在这条边时设为INF)

int d[MAX\_V];//顶点s出发的最短距离

bool used[MAX\_V];//已经使用过的图

int V;//顶点数

//求从起点s出发到各个顶点的最短距离

void dijkstra(int a)

{

fill(d,d+V,INF);

fill(used,used+V,false);

d[s]=0;

while(true)

{

int v=-1;

//从尚未使用过的顶点中选择一个距离最小的顶点

for(int u=0;u<V;u++)

{

if(!used[u]&&(v==-1||d[u]<d[v]))

v=u;

}

if(v==-1)

break;//没有找到就break

used[v]=true;

for(int u=0;u<V;u++)

{

d[u]=min(d[u],d[v]+cost[v][u]);

}

}

}

//使用邻接表算法的复杂度是 O(|E|log|V|)

//当所有边权值相等时，单源最短路可以通过BFS求解，此时，Dijkstra算法使用的

//priority\_queue和queue具有相同效果，因此复杂度会发生变化

struct edge

{

int to,cost;

edge(int x,int y){to=x;cost=y;}

};

typedef pair<int,int> P;//first是最短距离，second是顶点的编号

int V;

vector <edge>G[MAX\_V];

int d[MAX\_V];

void dijkstra(int s)

{

//通过指定greater<P>参数，堆按照first从小到大的顺序取出值

priority\_queue<P,vector<P>,greater<P> >que;

fill(d,d+V,INF);

d[s]=0;

que.push(P(0,s));

while(!que.empty())

{

P p=que.top();que.pop();

int v=p.second;

if(d[v]<p.first)

continue;

for(int i=0;i<G[v].size();i++)

{

edge e=G[v][i];

if(d[e.to]>d[v]+e.cost)

{

d[e.to]=d[v]+e.cost;

que.push(P(d[e.to],e.to));

}

}

}

}

/\*

相对于Bellman-Ford的O（|V||E|）的复杂度，Dijkstra算法的复杂度更高效

但仅适用于没有负边的情况

\*/

## 3、最短路 Floyd

/\*

任意两点间的最短路问题

（Floyd-Warshall算法）

在O(V^3)时间里求出所有两点间的最短路长度。

可以处理边是负数的情况，判断图中是否有负圈，只需检查是否存在

d[i][j]是负数的顶点i即可

\*/

int d[MAX\_V][MAX\_V];//d[u][v]表示边e=(u,v)的权值（不存在时设为INF，不过d[i][i]=0

int V;//顶点数

void warshall\_floyd()

{

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

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

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

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

}

## 4、最小生成树 Kruskal

/\*

kruskal最小生成树算法

按照边的权值的顺序从小到大看一遍，如果不产生圈（重边也考虑在内）

就把当前这条边加入到生成树中

是否产生圈只需看两点之前是否在同一连通分量里

使用并查集判断是否属于同一个连通分量

\*/

#define rank rankk //由于与某个库名称相同，故事先define

/\*

并查集

复杂度为阿克曼函数的反函数，比O(log(n))还快

\*/

int par[MAX];//父亲

int rank[MAX];//树的高度

//初始化n个元素

void init(int n)

{

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

{

par[i]=i;

rank[i]=0;

}

}

//查询树的根，期间加入了路径压缩

int find(int x)

{

if(par[x]==x)

return x;

else

return par[x]=find(par[x]);

}

//合并x和y所属的集合

void unite(int x,int y)

{

x=find(x);

y=find(y);

if(x==y)

return ;

if(rank[x]<rank[y])

par[x]=y;

else

{

par[y]=x;

if(rank[x]==rank[y])

rank[x]++;

}

}

//判断x和y是否属于同一个集合

bool same(int x,int y)

{

return find(x)==find(y);

}

/\*

建立结构体记录边

\*/

struct edge

{

int u,v,cost;

};

/\*储存边的数组\*/

edge es[MAX];

int V,E;//顶点数、边数

bool cmp(const edge &e1,const edge &e2)

{

return e1.cost<e2.cost;

}

ll kruskal(edge \*e,int edge\_num,int vertice\_num)//e为存储边的数组，下标从0开始

{

sort(e,e+edge\_num,cmp);//按照edge.cost顺序从小到大排序

init(vertice\_num);//初始化并查集

ll re=0;

for(int i=0;i<edge\_num;i++)

{

edge tem=e[i];

if(!same(tem.u,tem.v))

{

unite(tem.u,tem.v);

re+=(ll)tem.cost;

}

}

return re;

}

## 5、LCA

/\*

LCA 最近公共祖先

1、基于二分搜索的算法

\*/

//原始版 O(n) n为节点最大深度

vector<int> G[MAX\_V];//图的邻接表表示

int root;//根节点的编号

int parent[MAX\_V];//父亲节点（根节点父亲记为-1）

int depth[MAX\_V];//节点的深度

void dfs(int v,int p,int d)

{

parent[v]=p;

depth[v]=d;

for(int i=0;i<G[v].size();i++)

{

if(G[v][i]!=p)

dfs(G[v][i],v,d+1);

}

}

void init()

{ //预处理

dfs(root,-1,0);

}

int lca(int u,int v)

{

while(depth[u]>depth[v])

u=parent[u];

while(depth[v]>depth[u])

v=parent[v];

while(u!=v)

{

u=parent[u];v=parent[v];

}

return u;

}

//预处理 二分搜索版

//MAX\_LOG\_V 最少开到 log2(MAX\_V)+1

vector<int> G[MAX\_V];//图的邻接表表示

int root;//根节点编号

int parent[MAX\_LOG\_V][MAX\_V];//向上走2^k步时到达的节点 （超过根时记为-1）

int depth[MAX\_V];//节点的深度

void dfs(int v,int p,int d)

{

parent[0][v]=p;

depth[v]=d;

for(int i=0;i<G[v].size();i++)

{

if(G[v][i]!=p)

dfs(G[v][i],v,d+1);

}

}

//预处理

void init(int V)

{

//预处理出 parent[0]和depth

dfs(root,-1,0);

//预处理出parent

for(int k=0;k+1<MAX\_LOG\_V;k++)

for(int v=0;v<V;v++)

{

if(parent[k][v]<0)

parent[k+1][v]=-1;

else

parent[k+1][v]=parent[k][parent[k][v]];

}

}

//计算u和v的LCA

int lca(int u,int v)

{

//让u和v向上走到同一深度

if(depth[u]>depth[v])

swap(u,v);

for(int k=0;k<MAX\_LOG\_V;k++)

{

if(((depth[v]-depth[u])>>k)&1)

v=parent[k][v];

}

if(u==v)

return u;

for(int k=MAX\_LOG\_V-1;k>=0;k--)

{

if(parent[k][u]!=parent[k][v])

{

u=parent[k][u];v=parent[k][v];

}

}

return parent[0][u];

}

## 6、最大团

struct MAX\_CLIQUE {

static const int N=60;

bool G[N][N];

int n, Max[N], Alt[N][N], ans;

bool DFS(int cur, int tot) {

if(cur==0) {

if(tot>ans) {

ans=tot;

return 1;

}

return 0;

}

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

if(cur-i+tot<=ans) return 0;

int u=Alt[tot][i];

if(Max[u]+tot<=ans) return 0;

int nxt=0;

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

if(G[u][Alt[tot][j]]) Alt[tot+1][nxt++]=Alt[tot][j];

if(DFS(nxt, tot+1)) return 1;

}

return 0;

}

int MaxClique() {

ans=0, memset(Max, 0, sizeof Max);

for(int i=n-1; i>=0; i--) {

int cur=0;

for(int j=i+1; j<n; j++) if(G[i][j]) Alt[1][cur++]=j;

DFS(cur, 1);

Max[i]=ans;

}

return ans;

}

};

MAX\_CLIQUE edge;

int k;

int main() {

scanf("%d%d", &edge.n,&k);

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

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

scanf("%d", &edge.G[i][j]);

int da=edge.MaxClique();

printf("%.7f\n",(double)k\*k\*(da-1)/(2.0\*da));

return 0;

}

## 7、最大流（只保留Dinic）

//用于表示边的结构体（终点、容量、反向边）

struct edge{int to,cap,rev;};

const int MAX\_V=MAX;

vector <edge> G[MAX\_V];//图的邻接表表示

int level[MAX\_V];//顶点到源点的距离标号

int iter[MAX\_V];//当前弧 在其之前的边已经没有用了

//向图中增加一条从from到to的容量为cap的边

void add\_edge(int from,int to,int cap)

{

G[from].push\_back((edge){to,cap,G[to].size()});

G[to].push\_back((edge){from,0,G[to].size()-1});//反向边

}

//通过BFS计算从源点出发的距离标号

void bfs(int s)

{

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

queue<int>que;

level[s]=0;

que.push(s);

while(!que.empty())

{

int v=que.front();que.pop();

for(int i=0;i<G[v].size();i++)

{

edge &e=G[v][i];

if(e.cap>0&&level[e.to]<0)

{

level[e.to]=level[v]+1;

que.push(e.to);

}

}

}

}

//通过DFS寻找增广路

int dfs(int v,int t,int f)

{

if(v==t)return f;

for(int &i=iter[v];i<G[v].size();i++)

{

edge &e=G[v][i];

if(e.cap>0&&level[v]<level[e.to])

{

int d=dfs(e.to,t,min(f,e.cap));

if(d>0)

{

e.cap-=d;G[e.to][e.rev].cap+=d;//反向边增大

return d;

}

}

}

return 0;

}

int max\_flow(int s,int t)

{

int flow=0;

for(;;)

{

bfs(s);

if(level[t]<0)//已无能到t的增广路

return flow;

memset(iter,0,sizeof(iter));

int f;

while((f=dfs(s,t,INF))>0)

flow+=f;

}

}

## 8、二分图匹配

/\*

二分图匹配方法1：

加点（分别连向所有 “电脑” 从所有“任务”连出）作为源点和汇点

此时最大匹配（总点数）即为2\*最大流

\*/

//输入

int N,K;

bool can[MAX\_N][MAX\_K]; //can[i][j]=计算机i能够处理任务j

void solve()

{

//0-N-1 计算机对应的顶点

//N- N+K-1 任务对应的顶点

int s=N+K,t=S+1;//自行添加源点、汇点

//在源点和计算机之间连边

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

add\_edge(s,i,1);

//在任务和汇点之间连边

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

add\_edge(N+i,t,1);

//在计算机和任务之间连边

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

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

if(can[i][j])

add\_edge(i,n+j,1);

printf("%d\n",max\_flow(s,t));

}

/\*

算法二

利用所有边容量都是1以及二分图的性质

（匈牙利算法 DFS版）

\*/

int V;//顶点数

vector<int> G[MAX\_V];//图的邻接表表示

int match[MAX\_V];//所匹配的顶点

bool used[MAX\_V];//DFS中用到的访问标记

//向图中增加一条连接u和v的边

void add\_edge(int u,int v)

{

G[u].push\_back(v);

G[v].push\_back(u);

}

//通过DFS寻找增广路

bool dfs(int v)

{

used[v]=true;

for(int i=0;i<G[v].size();i++)

{

int u=G[v][i],w=match[u];

if(w<0||(!used[w]&&dfs(w)))

{

match[v]=u;

match[u]=v;

return true;

}

}

return false;

}

//求解二分图的最大匹配

int bipartite\_matching()

{

int res=0;

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

for(int v=0;v<V;v++)

{

if(match[v]<0)

{

memset(used,0,sizeof(used));

if(dfs(v))

++res;

}

}

return res;//返回的是总点数的一半

}

# 数据结构

## BIT

/\*

BIT

比起线段树，实现更简单，速度更快

i-=i&-i亦可写作 i=i&(i-1)

\*/

//[1,n]

int bit[MAX\_N+1],n;

int sum(int i)//[1,i]前缀和

{

int s=0;

while(i>0)

{

s+=bit[i];

i-=i&-i;

}

return s;

}

void add(int i,int x)//对给定的a[i],执行a[i]+=x

{

while(i<=n)

{

bit[i]+=x;

i+=i&-i;

}

}

## 并查集

见Kruskal、

# 数学

## FFT

struct cpx {

double x,y;

cpx(double a=0,double b=0):x(a),y(b) {}

};

cpx operator + (cpx a,cpx b) {return cpx(a.x+b.x,a.y+b.y);}

cpx operator - (cpx a,cpx b) {return cpx(a.x-b.x,a.y-b.y);}

cpx operator \* (cpx a,cpx b) {return cpx(a.x\*b.x-a.y\*b.y,a.x\*b.y+a.y\*b.x);}

const int maxn=500005;

const double pi=acos(-1.0);

char buf[maxn];

cpx A[maxn],B[maxn];

LL num[maxn];

void DFT(cpx\* a,int n,int d=1) {

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

if(i<j) swap(a[i],a[j]);

int k;for(k=(n>>1);i&k;i^=k,k>>=1);

i^=k;

}

for(int m=2;m<=n;m<<=1) {

cpx w=cpx(cos(pi\*2/m\*d),sin(pi\*2/m\*d));

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

cpx s=cpx(1,0);

for(int j=i;j<(i+(m>>1));j++) {

cpx u=a[j],v=s\*a[j+(m>>1)];

a[j]=u+v;a[j+(m>>1)]=u-v;

s=s\*w;

}

}

}

if(d==-1) for(int i=0;i<n;i++) a[i].x=a[i].x/n;

}

## 改进版FFT

#include <bits/stdc++.h>

using namespace std;

#define REP(i, a, b) for (int i = (a), \_end\_ = (b); i < \_end\_; ++i)

#define debug(...) fprintf(stderr, \_\_VA\_ARGS\_\_)

#define mp make\_pair

#define x first

#define y second

#define pb push\_back

#define SZ(x) (int((x).size()))

#define ALL(x) (x).begin(), (x).end()

template<typename T> inline bool chkmin(T &a, const T &b) { return a > b ? a = b, 1 : 0; }

template<typename T> inline bool chkmax(T &a, const T &b) { return a < b ? a = b, 1 : 0; }

typedef long long LL;

const int oo = 0x3f3f3f3f;

const int Mod = 1e9 + 7;

const int max0 = 262144;

struct comp

{

double x, y;

comp(): x(0), y(0) { }

comp(const double &\_x, const double &\_y): x(\_x), y(\_y) { }

};

inline comp operator+(const comp &a, const comp &b) { return comp(a.x + b.x, a.y + b.y); }

inline comp operator-(const comp &a, const comp &b) { return comp(a.x - b.x, a.y - b.y); }

inline comp operator\*(const comp &a, const comp &b) { return comp(a.x \* b.x - a.y \* b.y, a.x \* b.y + a.y \* b.x); }

inline comp conj(const comp &a) { return comp(a.x, -a.y); }

const double PI = acos(-1);

int N, L;

comp w[max0 + 5];

int bitrev[max0 + 5];

void fft(comp \*a, const int &n)

{

REP(i, 0, n) if (i < bitrev[i]) swap(a[i], a[bitrev[i]]);

for (int i = 2, lyc = n >> 1; i <= n; i <<= 1, lyc >>= 1)

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

{

comp \*l = a + j, \*r = a + j + (i >> 1), \*p = w;

REP(k, 0, i >> 1)

{

comp tmp = \*r \* \*p;

\*r = \*l - tmp, \*l = \*l + tmp;

++l, ++r, p += lyc;

}

}

}

inline void fft\_prepare()

{

REP(i, 0, N) bitrev[i] = bitrev[i >> 1] >> 1 | ((i & 1) << (L - 1));//FFTµÄ01´®

REP(i, 0, N) w[i] = comp(cos(2 \* PI \* i / N), sin(2 \* PI \* i / N));

}

inline void conv(int \*x, int \*y, int \*z)

{

REP(i, 0, N) (x[i] += Mod) %= Mod, (y[i] += Mod) %= Mod;

static comp a[max0 + 5], b[max0 + 5];

static comp dfta[max0 + 5], dftb[max0 + 5], dftc[max0 + 5], dftd[max0 + 5];

REP(i, 0, N) a[i] = comp(x[i] & 32767, x[i] >> 15);

REP(i, 0, N) b[i] = comp(y[i] & 32767, y[i] >> 15);

fft(a, N), fft(b, N);

REP(i, 0, N)

{

int j = (N - i) & (N - 1);

static comp da, db, dc, dd;

da = (a[i] + conj(a[j])) \* comp(0.5, 0);

db = (a[i] - conj(a[j])) \* comp(0, -0.5);

dc = (b[i] + conj(b[j])) \* comp(0.5, 0);

dd = (b[i] - conj(b[j])) \* comp(0, -0.5);

dfta[j] = da \* dc;

dftb[j] = da \* dd;

dftc[j] = db \* dc;

dftd[j] = db \* dd;

}

REP(i, 0, N) a[i] = dfta[i] + dftb[i] \* comp(0, 1);

REP(i, 0, N) b[i] = dftc[i] + dftd[i] \* comp(0, 1);

fft(a, N), fft(b, N);

REP(i, 0, N)

{

int da = (LL)(a[i].x / N + 0.5) % Mod;

int db = (LL)(a[i].y / N + 0.5) % Mod;

int dc = (LL)(b[i].x / N + 0.5) % Mod;

int dd = (LL)(b[i].y / N + 0.5) % Mod;

z[i] = (da + ((LL)(db + dc) << 15) + ((LL)dd << 30)) % Mod;

}

}

int main()

{

#ifndef ONLINE\_JUDGE

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

freopen("output.txt", "w", stdout);

#endif

int n, m;

static int a[max0 + 5], b[max0 + 5], c[max0 + 5];

scanf("%d%d", &n, &m), ++n, ++m;

REP(i, 0, n) scanf("%d", a + i);

REP(i, 0, m) scanf("%d", b + i);

L = 0;

for ( ; (1 << L) < n + m - 1; ++L);

N = 1 << L;//NÏàµ±ÓÚ1<<L

fft\_prepare();

conv(a, b, c);

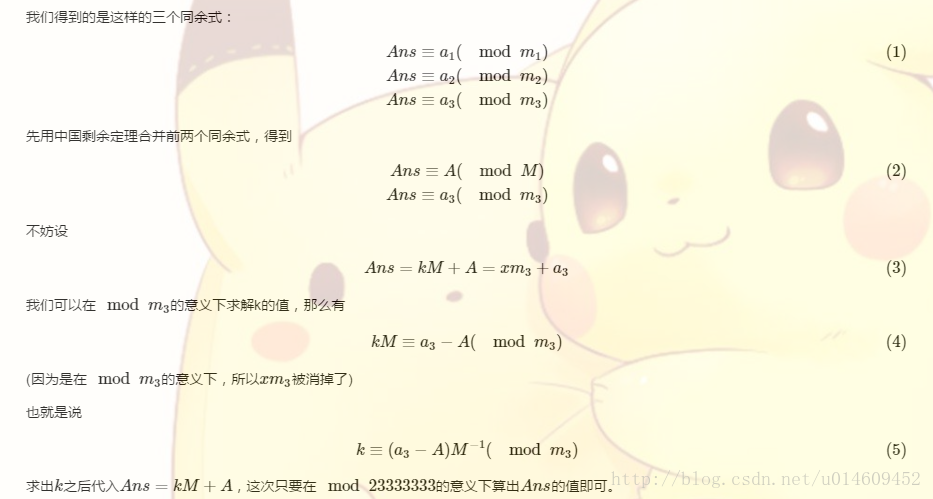
REP(i, 0, n + m - 1) (c[i] += Mod) %= Mod, printf("%d ", c[i]);

printf("\n");

return 0;

}

## NTT 及模任意数



inline ll mul(ll a,ll b,ll p){

a%=p; b%=p;

return ((a\*b-(ll)((ll)((long double)a/p\*b+1e-3)\*p))%p+p)%p;

}

ll euler[maxn+5];

void getEuler()

{

memset(euler,0,sizeof(euler));

euler[1] = 1;

for(int i = 2;i <= (int)1e5;i++)

if(!euler[i])

for(int j = i;j <= (int)1e5; j += i)

{

if(!euler[j]) euler[j] = j;

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

}

}

/\*NTT部分\*/

//const int N = 1 << 18;

const int NUM = 20;

ll wn[5][NUM];

ll P[5]={469762049,998244353,1004535809};

ll fac[maxn],inv[maxn];

ll G=3;

ll A[3][maxn<<1],B[maxn<<1];

ll quick\_mod(ll a, ll b, ll m)

{

ll ans = 1;

a %= m;

while(b)

{

if(b & 1)

{

ans = ans \* a % m;

b--;

}

b >>= 1;

a = a \* a % m;

}

return ans;

}

void GetWn(int id)//预处理原根的幂次

{

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

{

int t = 1 << i;

wn[id][i] = quick\_mod(G, (P[id] - 1) / t, P[id]);

}

}

void Rader(ll a[], int len)

{

int j = len >> 1;

for(int i = 1; i < len - 1; i++)

{

if(i < j) swap(a[i], a[j]);

int k = len >> 1;

while(j >= k)

{

j -= k;

k >>= 1;

}

if(j < k) j += k;

}

}

void NTT(ll a[], int len, int ids,int on=1)//NTT的数组 下标从0开始 数组长度len

{

Rader(a, len);

int id = 0;

for(int h = 2; h <= len; h <<= 1)

{

id++;

for(int j = 0; j < len; j += h)

{

ll w = 1;

for(int k = j; k < j + h / 2; k++)

{

ll u = a[k] % P[ids];

ll t = w \* a[k + h / 2] % P[ids];

a[k] = (u + t) % P[ids];

a[k + h / 2] = (u - t + P[ids]) % P[ids];

w = w \* wn[ids][id] % P[ids];

}

}

}

if(on == -1)

{

for(int i = 1; i < len / 2; i++)

swap(a[i], a[len - i]);

ll inv = quick\_mod(len, P[ids] - 2, P[ids]);

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

a[i] = a[i] \* inv % P[ids];

}

}

int t;

int n;

ll mo,an;

ll crt(ll x,ll y,ll z)

{

ll M=P[0]\*P[1];

ll inv1=quick\_mod(P[0]%P[1],P[1]-2LL,P[1]),inv2=quick\_mod(P[1]%P[0],P[0]-2LL,P[0]),inv12=quick\_mod(M%P[2],P[2]-2LL,P[2]);

ll re=(mul(x\*P[1]%M,inv2,M)+mul(y\*P[0]%M,inv1,M))%M;

ll k=(z+P[2]-re%P[2])%P[2]\*inv12%P[2];

return (k\*(M%mo)%mo+re)%mo;

}

int main() //例一

{

for(int i=0;i<3;i++)

GetWn(i);

getEuler();

scanf("%d",&t);

while(t--)

{

scanf("%d%lld",&n,&mo);

P[3]=mo;

fac[0]=1LL;

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

fac[j]=fac[j-1]\*(ll)j%mo;

inv[n]=quick\_mod(fac[n],mo-2,mo);

for(int j=n-1;j>=0;j--)

inv[j]=(ll)(j+1LL)\*inv[j+1]%mo;

an=0;

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

{

int ge=n/g;

ll temhe=0;

if(ge<2)

break;

memset(A,0,sizeof(A));

int length=0;

while((1<<length)<2\*ge)

++length;

for(int j=0;j<3;j++)

{

memset(B,0,sizeof(B));

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

A[j][i]=B[i]=inv[(i+1)\*g];

NTT(A[j],1<<length,j);NTT(B,1<<length,j);

for(int i=0;i<(1<<length);i++)

A[j][i]=A[j][i]\*B[i]%P[j];

NTT(A[j],1<<length,j,-1);

}

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

{

temhe=(temhe+mul(crt(A[0][i],A[1][i],A[2][i]),inv[n-(i+2)\*g],mo))%mo;

}

temhe=mul(temhe,fac[n],mo);

an=(an+mul(temhe,(ll)euler[g],mo))%mo;

}

an=(an+(ll)n\*powMM(2LL,(ll)n,mo)%mo)%mo;

an=an\*powMM(3LL,(ll)n,mo)%mo;

printf("%lld\n",an);

}

}

int main() //例二

{

GetWn();//预处理原根的幂次

while(scanf("%s %s", A, B) != EOF)

{

int len;

Prepare(A, B, a, b, len);//将字符串形式转化

Conv(a, b, len);//多项式乘法 NTT 与其还原

Transfer(a, len);//进位转化

Print(a, len);//输出

}

return 0;

}

## 二次剩余

#define ull unsigned long long

#define ll long long

#define N 100005

namespace random\_gen{

ull r=0x1234567890ABCDEF;

ull f(){

r=r\*0x234567890FEDBCA1+0xABCFED0987654123;

return r=r>>29|r<<35;

}

}

namespace numbertheory{

ll mulmod(ll a,ll b,ll p){

ll s=0;

while(b){

if(b&1) s=(s+a)%p;

b>>=1;a=(a+a)%p;

}

return s;

}

ll powmod(ll a,ll b,ll p){

ll s=1;

while(b){

if(b&1) s=mulmod(s,a,p);

b>>=1;a=mulmod(a,a,p);

}

return s;

}

ll Legendre(ll a,ll p){

return powmod(a,p-1>>1,p);

}

namespace fieldextension{

ll d,p;

struct num{

ll x,y;

};

num operator\*(num a,num b){

return (num){(mulmod(a.x,b.x,p)+mulmod(mulmod(a.y,b.y,p),d,p))%p,(mulmod(a.x,b.y,p)+mulmod(a.y,b.x,p))%p};

}

num operator^(num a,ll b){

num s=(num){1,0};

while(b){

if(b&1) s=s\*a;

b>>=1;a=a\*a;

}

return s;

}

}

ll quadratic\_equation(ll n,ll p){

if(!n) return 0;

if(p==2) return 1;

if(Legendre(n,p)==p-1) return -1;

ll a,w;

while(1){

a=random\_gen::f()%p;

w=(mulmod(a,a,p)-n+p)%p;

//printf("%lld %lld\n",a,w);

if(Legendre(w,p)==p-1){

fieldextension::d=w;

fieldextension::p=p;

return ((fieldextension::num){a,1}^p+1>>1).x;

}

}

}

}

## 逆元及求组合数

const ll MOD=1e9+7;

const int MAX=1e5+5;

/\*

fi数组储存阶乘

inv数组储存逆元

计算逆元有两种办法

方法1：先求出最大的一个的逆元（用费马小定理）

之后倒序递推 inv[i]=(i+1)inv[i+1] (两侧同时乘i!显然成立)

方法2：……没看懂

\*/

ll fi[MAX],inv[MAX];

ll quick(ll a,ll b)

{

ll an=1LL;

while(b)

{

if(b&1)

{

an=an\*a%MOD;

}

a=a\*a%MOD;

b/=2;

}

return an%MOD;

}

ll C(ll a,ll b)

{

if(a<0||b<0||b>a)

return 0LL;

return (fi[a]\*inv[b]%MOD)\*inv[a-b]%MOD;

}

int n,t,w;

ll ans,x,c;

int main()

{

for(ll i=0;i<=1e5;i++)

{

fi[i]=i?(fi[i-1]\*i%MOD):1;

}

/\*计算逆元的第一种方法\*/

inv[100000]=quick(fi[100000],MOD-2);

for(ll i=1e5-1;i>=0;i--)

inv[i]=(i+1)\*inv[i+1]%MOD;

/\*计算逆元的第二种方法\*/

inv[1]=inv[0]=1;

for (int i=2;i<=100000;i++)

{

inv[i]=(LL)(mod-mod/i)\*inv[mod%i]%mod;

}

for (int i=2;i<=100000;i++)

{

inv[i]=(inv[i]\*inv[i-1])%mod;

}

}

/\*

求对整数（非阶乘）的模P下的逆元

\*/

for(inv[1]=1,i=2;i<P;i++)

inv[i]=1LL\*(P-inv[P%i])\*(P/i)%P;

## 同种不相邻的排列数

（大复杂度版）

ll fi[MAX],inv[MAX];

ll quick(ll a,ll b)

{

ll an=1LL;

while(b)

{

if(b&1)

{

an=an\*a%MOD;

}

a=a\*a%MOD;

b/=2;

}

return an%MOD;

}

ll C(ll a,ll b)

{

if(a<0||b<0||b>a)

return 0LL;

return (fi[a]\*inv[b]%MOD)\*inv[a-b]%MOD;

}

bool check(ll x,ll y)

{

ll ji=x\*y;

ll sq=(ll)sqrt(ji);

if(sq\*sq==ji)

return true;

return false;

}

int n;

int a[MAX];

bool vi[MAX];

ll per;

queue<int>que;

ll dp[305][305];

int main()

{

for(ll i=0;i<=300;i++)

{

fi[i]=i?(fi[i-1]\*i%MOD):1;

}

inv[300]=quick(fi[300],MOD-2);

for(ll i=299;i>=0;i--)

inv[i]=(i+1)\*inv[i+1]%MOD;

scanf("%d",&n);per=1;

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

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

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

{

if(vi[i])

continue;

int cnt=1;

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

{

if(check(a[i],a[j]))

{

++cnt;vi[j]=true;

}

}

per=per\*fi[cnt]%MOD;

que.push(cnt);

}

int tot=que.size();

int num=que.front();que.pop();

int lin=num-1,id=1;

dp[1][num-1]=1;

while(!que.empty())

{

num=que.front();que.pop();++id;

for(int i=0;i<=lin;i++)//原本有i个不合法空隙

{

for(int j=1;j<=num;j++)//将这种数分成几块

{

// dp[id][]

for(int s=0;s<=i&&s<=j;s++)//破坏多少个原本的空隙

dp[id][i-s+num-j]=(dp[id][i-s+num-j]+dp[id-1][i]\*C(num-1,j-1)%MOD\*C(i,s)%MOD\*C(id+lin-i,j-s)%MOD)%MOD;

}

}

lin+=num-1;

}

printf("%I64d\n",dp[tot][0]\*per%MOD);

}

（小复杂度版）

ll fi[MAX],inv[MAX];

ll quick(ll a,ll b)

{

ll an=1LL;

while(b)

{

if(b&1)

{

an=an\*a%MOD;

}

a=a\*a%MOD;

b/=2;

}

return an%MOD;

}

ll C(ll a,ll b)

{

if(a<0||b<0||b>a)

return 0LL;

return (fi[a]\*inv[b]%MOD)\*inv[a-b]%MOD;

}

int a[MAX];

ll dp[2][MAX],an;//总共分成j组的分法数

int t,sum,now;

int main()

{

for(ll i=0;i<=4005;i++)

{

fi[i]=i?(fi[i-1]\*i%MOD):1;

}

inv[4005]=quick(fi[4005],MOD-2);

for(ll i=4004;i>=0;i--)

inv[i]=(i+1)\*inv[i+1]%MOD;

while(~scanf("%d%d%d%d",&a[1],&a[2],&a[3],&a[4]))

{

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

now^=1;sum=0;dp[now][0]=1;

an=0;

for(int i=1;i<=4;i++)

{

if(a[i])

{

for(int j=0;j<=sum;j++)//已分成的组数

{

for(int s=1;s<=a[i];s++)//当前种类分成的组数

{

dp[now^1][j+s]=(dp[now^1][j+s]+dp[now][j]\*C(a[i]-1,s-1)%MOD\*inv[s]%MOD)%MOD;

}

}

sum+=a[i];memset(dp[now],0,sizeof(dp[now]));now^=1;

}

}

if(!sum)

{

printf("0\n");continue;

}

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

{

ll tem=dp[now][sum-i];

if(i&1)

tem=MOD-tem;

an=(an+tem\*fi[sum-i]%MOD)%MOD;

}

printf("%lld\n",an);

}

return 0;

}

## 原根

const int N = 1000010;

bitset<N> prime;

int p[N],pri[N];

int k,cnt;

void isprime()

{

prime.set();

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

{

if(prime[i])

{

p[k++] = i;

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

prime[j] = false;

}

}

}

void Divide(int n)

{

cnt = 0;

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

for(int i=0; p[i]<=t; i++)

{

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

{

pri[cnt++] = p[i];

while(n%p[i]==0) n /= p[i];

}

}

if(n > 1)

pri[cnt++] = n;

}

ll quick\_mod(ll a,ll b,ll m)

{

ll ans = 1;

a %= m;

while(b)

{

if(b&1)

{

ans = ans \* a % m;

b--;

}

b >>= 1;

a = a \* a % m;

}

return ans;

}

int main()

{

int P;

isprime();

while(cin>>P)

{

Divide(P-1);

for(int g=2; g<P; g++)

{

bool flag = true;

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

{

int t = (P - 1) / pri[i];

if(quick\_mod(g,t,P) == 1)

{

flag = false;

break;

}

}

if(flag)

{

int root = g;

cout<<root<<endl;

system("pause");

}

}

}

return 0;

}

## 中国剩余定理

int CRT(int a[],int m[],int n)

{

int M = 1;

int ans = 0;

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

M \*= m[i];

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

{

int x, y;

int Mi = M / m[i];

extend\_Euclid(Mi, m[i], x, y);

ans = (ans + Mi \* x \* a[i]) % M;

}

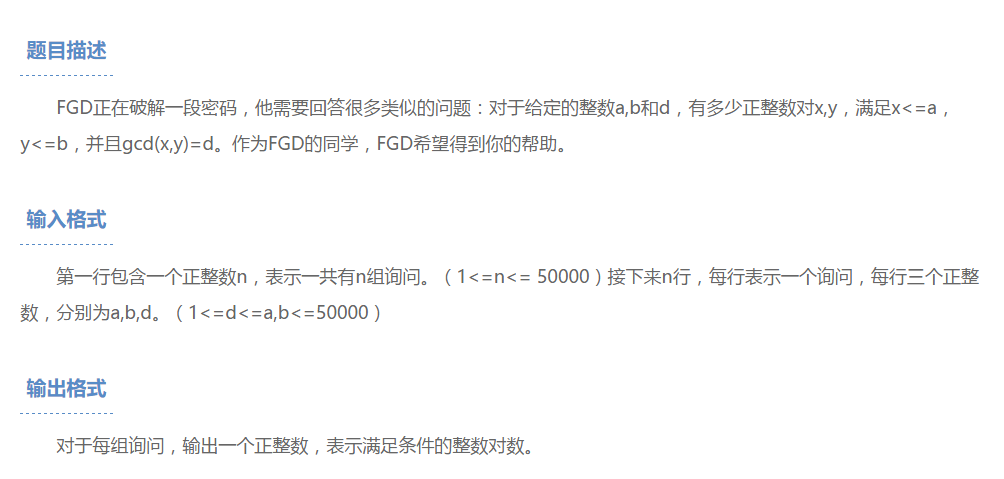
if(ans < 0) ans += M;

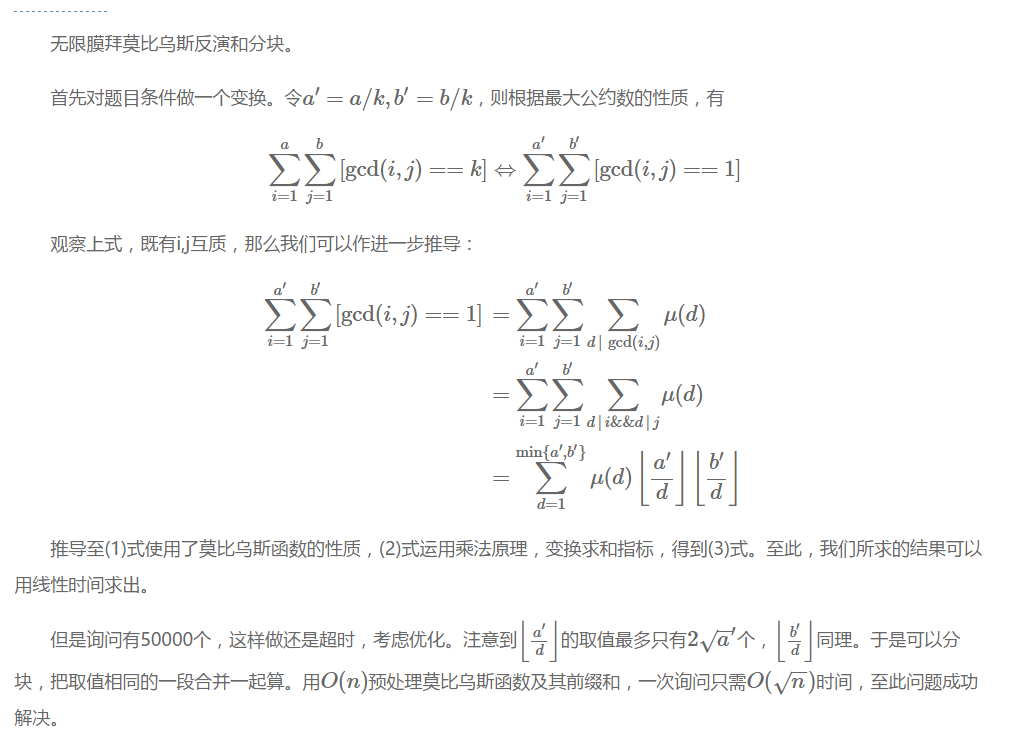
return ans;

}

## 莫比乌斯反演几例

例一





例二

根据莫比乌斯反演可得

应用到本题的式子中