MST

struct edge

{

int u,v,w;

bool operator < ( const edge& p ) const

{

return w < p.w;

}

};

int pr[MAXN];

vector<edge>e;

int find(int r)

{

return (pr[r]==r) ? r: pr[r]=find(pr[r]);

}

int mst(int n)

{

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

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

int count=0,s=0;

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

{

int u=find(e[i].u);

int v=find(e[i].v);

if(u!=v)

{

pr[u]=v;

count++;

s+=e[i].w;

if(count=K=n-1) break;

}

}

return s;

}

scc

#define graph\_size 100001

vector<int> G[graph\_size],RG[graph\_size];

vector<int> components[graph\_size];

stack<int> st;

bool vis[graph\_size];

int mark;

void dfs(int u)

{

vis[u]=1;

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

int v=G[u][i];

if(!vis[v]) dfs(v);

}

st.push(u);

return ;

}

void dfs2(int u,int mark){

components[mark].push\_back(u);

vis[u]=1;

for(int i=0;i<RG[u].size();i++){

int v=RG[u][i];

if(!vis[v]) dfs2(v,mark);

}

return ;

}

void SCC(int n){

while(!st.empty()) st.pop();

for(int i=0;i<=n;i++) components[i].clear();

mark=0;

memset(vis,0,sizeof vis);

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

if(!vis[i]) dfs(i);

}

memset(vis,0,sizeof vis);

while(!st.empty()){

int u=st.top();

st.pop();

if(!vis[u]){

dfs2(u,mark);

mark++;

}

}

}

KMP

int f[1000007];

void failure\_function(char \*pattern)

{

f[0] = 0;

int k = 1, len = 0, len\_p = strlen(pattern);

while (k < len\_p)

{

if (pattern[k] == pattern[len])f[k++] = ++len;

else

{

if (len)len = f[len - 1];

else f[k++] = 0;

}

}

return;

}

void KMP\_match(char \*txt, char\*pattern)

{

int i = 0, j = 0, ret = -1;

int len\_t = strlen(txt), len\_p = strlen(pattern);

while (i < len\_t)

{

if (txt[i] == pattern[j])

{

i++; j++;

if (j == len\_p)

{

ret = i - len\_p;

printf("A match found from index %d\n", ret);

j = f[j - 1];

}

}

else

{

if (j)j = f[j - 1];

else i++;

}

}

}

int main()

{

int tc, t = 0;

scanf("%d\n", &tc);

while (tc--)

{

char TXT[ 1000007], PT[1000007];

gets(TXT);

gets(PT);

failure\_function(PT);

KMP\_match(TXT, PT);

}

}

MATRIX:

template<int N> class matrix {

public:

int arr[N][N];

matrix() {

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

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

arr[i][j] = 0 ;

}

}

}

matrix<N> operator \*(const matrix<N> &in) {

matrix<N> ret ;

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

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

for( int k = 0 ; k < N ; k++ ) {

ret.arr[i][j]+=(arr[i][k])\*(in.arr[k][j]) ;

ret.arr[i][j]%=10000 ;

}

}

return ret ;

}

matrix<N> operator ^( int POW ) {

matrix<N> ret ;

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

ret.arr[i][i] = 1 ;

}

matrix<N> ME = \*this ;

while( POW ) {

if( POW&1 ) {

ret = ret \* ME ;

}

ME = ME \* ME ;

POW >>= 1 ;

}

return ret ;

}

};

BIGMOD

int bigmod(long long B,long long P,long long MOD)

{

long long R=1;

while(P>0){

if(P%2==1){

R=(R\*B)%MOD;

}

P/=2;

B=(B\*B)%MOD;

}

return R;

}

GENERATE DIVISORS

#define SIZE\_N 100

#define SIZE\_P 100

bool flag[SIZE\_N+5];

int primes[SIZE\_P+5];

int seive()

{

int i,j,total=0,val;

for(i=2; i<=SIZE\_N; i++) flag[i]=1;

val=sqrt(SIZE\_N)+1;

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

if(flag[i]){

for(j=i; j\*i<=SIZE\_N; j++) flag[i\*j]=0;

}

}

for(i=2; i<=SIZE\_N; i++){

if(flag[i]) primes[total++]=i;

}

return total;

}

int store\_primes[100],freq\_primes[100], store\_divisor[10000], Total\_Prime, ans;

void divisor(int N)

{

int i,val,ct;

val=sqrt(N)+1;

Total\_Prime=0;

for(i=0; primes[i]<val; i++){

if(N%primes[i]==0){

ct=0;

while(N%primes[i]==0){

N/=primes[i];

ct++;

}

store\_primes[Total\_Prime]=primes[i];

freq\_primes[Total\_Prime]=ct;

Total\_Prime++;

val=sqrt(N)+1;

}

}

if(N>1){

store\_primes[Total\_Prime]=N;

freq\_primes[Total\_Prime]=1;

Total\_Prime++;

}

}

void Generate(int cur,int num)

{

int i,val;

if(cur==Total\_Prime){

store\_divisor[ans++]=num;

}

else

{

val=1;

for(i=0; i<=freq\_primes[cur]; i++){

Generate(cur+1,num\*val);

val=val\*store\_primes[cur];

}

}

}

int main()

{

int total=seive();

int n,i;

while(scanf("%d",&n)==1){

divisor(n);

ans=0;

Generate(0,1);

sort(&store\_divisor[0],&store\_divisor[ans]);

printf("Total No of Divisors: %d\n",ans);

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

printf("%d ",store\_divisor[i]);

}

printf("\n");

}

return 0;

}

NCR

#define size\_nCr 1001

ll int combination[size\_nCr][size\_nCr];

void precal\_nCr()

{

combination[0][0]=1;

for(int i=0;i<size\_nCr;i++){

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

if(j==i || j==0) combination[i][j]=1;

else if(j==1)combination[i][j]=i;

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

}

}

}

ll int nCr(int n, int r)

{

if(n>=size\_nCr || r>=size\_nCr) return 0;

if(n<0 || r<0) return 0;

else return Kcombination[n][r];

}

PHI FUNCTION

int phi[1000105];

void PHI(){

phi[1]=0;

for(int i=2;i<=1000100;i++)phi[i]=i;

for(int i=2;i<=1000100;i++){

if(phi[i]==i){

phi[i]--;

for(int j=i+i;j<=1000100;j+=i){

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

}

}

}

}

SUM OF DIVISROS

SEIVE FIRST

int SOD(int N)

{

int i,val,sum,p,s;

val=sqrt(N)+1;

sum=1;

for(i=0; primes[i]<val; i++){

if(N%primes[i]==0){

p=1;

while(N%primes[i]==0){

N/=primes[i];

p=p\*primes[i];

}

p=p\*primes[i];

s=(p-1)/(primes[i]-1);

sum=sum\*s;

}

}

if(N>1){

p=N\*N;

s=(p-1)/(N-1);

sum=sum\*s;

}

return sum;

}

/\*

ConvexHull : Graham's Scan O(n lg n), integer implementation

P[]: holds all the points, C[]: holds points on the hull

np: number of points in P[], nc: number of points in C[]

to handle duplicate, call makeUnique() before calling convexHull()

call convexHull() if you have np >= 3

to remove co-linear points on hull, call compress() after convexHull()

\*/

point P[MAX], C[MAX], P0;

inline int triArea2(const point &a, const point &b, const point &c) {

return (a.x\*(b.y-c.y) + b.x\*(c.y-a.y) + c.x\*(a.y-b.y));

}

inline int sqDist(const point &a, const point &b) {

return ((a.x-b.x)\*(a.x-b.x) + (a.y-b.y)\*(a.y-b.y));

}

inline bool comp(const point &a, const point &b) {

int d = triArea2(P0, a, b);

if(d < 0) return false;

if(!d && sqDist(P0, a) > sqDist(P0, b)) return false;

return true;

}

inline bool normal(const point &a, const point &b) {

return ((a.x==b.x) ? a.y < b.y : a.x < b.x);

}

inline bool issame(const point &a, const point &b) {

return (a.x == b.x && a.y == b.y);

}

inline void makeUnique(int &np) {

sort(&P[0], &P[np], normal);

np = unique(&P[0], &P[np], issame) - P;

}

void convexHull(int &np, int &nc) {

int i, j, pos = 0;

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

if(P[i].y<P[pos].y || (P[i].y==P[pos].y && P[i].x<P[pos].x))

pos = i;

swap(P[0], P[pos]);

P0 = P[0];

sort(&P[1], &P[np], comp);

for(i = 0; i < 3; i++) C[i] = P[i];

for(i = j = 3; i < np; i++) {

while(triArea2(C[j-2], C[j-1], P[i]) < 0) j--;

C[j++] = P[i];

}

nc = j;

}

void compress(int &nc) {

int i, j, d;

C[nc] = C[0];

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

d = triArea2(C[j-1], C[i], C[i+1]);

if(d || (!d && issame(C[j-1], C[i+1]))) C[j++] = C[i];

}

nc = j;

}

/\*

C[] array of points of convex polygon in ccw order,

nc number of points in C, p target points.

returns true if p is inside C (including edge) or false otherwise.

complexity O(lg n)

\*/

inline bool inConvexPoly(point \*C, int nc, const point &p) {

int st = 1, en = nc - 1, mid;

while(en - st > 1) {

mid = (st + en)>>1;

if(triArea2(C[0], C[mid], p) < 0) en = mid;

else st = mid;

}

if(triArea2(C[0], C[st], p) < 0) return false;

if(triArea2(C[st], C[en], p) < 0) return false;

if(triArea2(C[en], C[0], p) < 0) return false;

return true;

}

/\*

P[] holds the points, must be either in cw or ccw

function returns double of the area.

\*/

inline int dArea(int np) {

int area = 0;

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

area += p[i].x\*p[i+1].y - p[i].y\*p[i+1].x;

}

return abs(area);

}

/\*

This code assumes the circle center and radius to be integer.

Change this when necessary.

\*/

inline double commonArea(const Circle &a, const Circle &b) {

int dsq = sqDist(a.c, b.c);

double d = sqrt((double)dsq);

if(sq(a.r + b.r) <= dsq) return 0;

if(a.r >= b.r && sq(a.r-b.r) >= dsq) return pi \* b.r \* b.r;

if(a.r <= b.r && sq(b.r-a.r) >= dsq) return pi \* a.r \* a.r;

double angleA = 2.0 \* acos((a.r \* a.r + dsq - b.r \* b.r) / (2.0 \* a.r \* d));

double angleB = 2.0 \* acos((b.r \* b.r + dsq - a.r \* a.r) / (2.0 \* b.r \* d));

return 0.5 \* (a.r \* a.r \* (angleA - sin(angleA)) + b.r \* b.r \* (angleB - sin(angleB)));

}

/\*

Segment intersection in 2D integer space.

P1, p2 makes first segment, p3, p4 makes the second segment

\*/

inline bool intersect(const Point &p1, const Point &p2, const Point &p3, const Point &p4) {

i64 d1, d2, d3, d4;

d1 = direction(p3, p4, p1);

d2 = direction(p3, p4, p2);

d3 = direction(p1, p2, p3);

d4 = direction(p1, p2, p4);

if(((d1 < 0 && d2 > 0) || (d1 > 0 && d2 < 0)) && ((d3 < 0 && d4 > 0) || (d3 > 0 && d4 < 0))) return true;

if(!d3 && onsegment(p1, p2, p3)) return true;

if(!d4 && onsegment(p1, p2, p4)) return true;

if(!d1 && onsegment(p3, p4, p1)) return true;

if(!d2 && onsegment(p3, p4, p2)) return true;

return false;

}

/\*

Some tetrahedron formulas

\*/

inline double volume(double u, double v, double w, double U, double V, double W) {

double u1,v1,w1;

u1 = v \* v + w \* w - U \* U;

v1 = w \* w + u \* u - V \* V;

w1 = u \* u + v \* v - W \* W;

return sqrt(4.0\*u\*u\*v\*v\*w\*w - u\*u\*u1\*u1 - v\*v\*v1\*v1 - w\*w\*w1\*w1 + u1\*v1\*w1) / 12.0;

}

inline double surface(double a, double b, double c) {

return sqrt((a + b + c) \* (-a + b + c) \* (a - b + c) \* (a + b - c)) / 4.0;

}

inline double insphere(double WX, double WY, double WZ, double XY, double XZ, double YZ) {

double sur, rad;

sur = surface(WX, WY, XY) + surface(WX, XZ, WZ) + surface(WY, YZ, WZ) + surface(XY, XZ, YZ);

rad = volume(WX, WY, WZ, YZ, XZ, XY) \* 3.0 / sur;

return rad;

}

/\*

1. Assign hull.n

2. Take input in hull.point

3. Call convexSort()

4. Call findHull()

5. Convex Hull is ready in hull.convex with hull.m points in it.

\*/

pll g;

vlong triArea ( pll a, pll b, pll c ) {

vlong area = a.ff \* b.ss + b.ff \* c.ss + c.ff \* a.ss;

area -= a.ff \* c.ss + b.ff \* a.ss + c.ff \* b.ss;

return area;

}

vlong sqDist ( pll a, pll b ) {

return ( SQ(a.ff-b.ff) + SQ(a.ss-b.ss ) );

}

bool convexCompare ( const pll &a, const pll &b ) {

vlong area = triArea ( g, a, b );

if ( area > 0 ) return true;

else if ( area == 0 && sqDist ( g, a ) < sqDist ( g, b ) ) return true;

else return false;

}

struct ConvexHull {

int n, m;

pll point[PPP], convex[PPP];

void convexSort() {

g = point[0];

FOR(i,0,n-1) {

if ( point[i].ff < g.ff ) g = point[i];

else if ( point[i].ff == g.ff && point[i].ss < g.ss ) g = point[i];

}

sort ( point, point + n, convexCompare );

}

void findHull() {

if ( n == 1 ) {

convex[0] = convex[1] = point[0];

m = 1;

return;

}

convex[0] = point[n-1]; convex[1] = point[0]; convex[2] = point[1];

int cur = 3;

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

vlong area = triArea ( convex[cur-2], convex[cur-1], point[i] );

if ( area > 0 ) {

convex[cur] = point[i];

cur++;

}

else if ( area == 0 ) { ///Take action depending on what is required

/\*Left Vertical Line gets omitted. Manually handle it\*/

/\*convex[cur] = point[i];

cur++;\*/

///If extra point needs to be removed

convex[cur-1] = point[i];

}

else {

cur--;

i--;

}

}

m = cur - 1;

}

}hull;

Monotone Chain

#include<bits/stdc++.h>

#define pi acos(-1.0)

using namespace std ;

struct Point {

long long x, y;

int idx ;

bool operator <(const Point &p) const {

return x < p.x || (x == p.x && y < p.y);

}

};

int n ;double d ;vector<Point> arr ;

long long cross(const Point &O, const Point &A, const Point &B){

return (A.x - O.x) \* (B.y - O.y) - (A.y - O.y) \* (B.x - O.x);

}

vector<Point> convex\_hull(vector<Point> P)

{

int n = P.size(), k = 0;

vector<Point> H(2\*n);

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

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

while (k >= 2 && cross(H[k-2], H[k-1], P[i]) < 0) k--;

H[k++] = P[i];

}

for (int i = n-2, t = k+1; i >= 0; i--) {

while (k >= t && cross(H[k-2], H[k-1], P[i]) < 0) k--;

H[k++] = P[i];

}

H.resize(k);

return H;

}