//

// math.h

// DailyCodingTeamNote

//

// Created by IRIS on 9/7/15.

// Copyright (c) 2015 IRIS. All rights reserved.

//

#include <stdio.h>

#include <math.h>

#include <vector>

using namespace std;

//Complexity : O(N/ logN + N ^ 0.75) for worst case (which means when n is prime number)

// N= 10^9 -> 5 \* 10^7

// N= 10^10 -> 4.6 \* 10^8

// N= 10^11 -> 4.1 \* 10^9

bool isPrime(int n) {

return isPrime(n, getPrimes((int)sqrt(n)));

}

//Complexity : O(N) for worst case (which means when n is prime number)

bool isPrime(int n, const vector<int> v) {

for (auto now : v) {

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

}

return true;

}

//Verified in range of (0, 10^6) at least by BOJ

//Complexity : O(N ^1.5)

vector<int> getPrimes(int N) {

vector<int> ret;

if (N >= 2) ret.push\_back(2);

if (N >= 3) ret.push\_back(3);

int i, j, k;

bool ctn = true;

int mid\_point = (int)sqrt(N - 1) / 6 + 1;

for (i = 1; ctn && i <= mid\_point; i++) {

for (j = -1; j <= 1; j += 2) {

int now = i \* 6 + j;

if (now > sqrt(N)) {

ctn = false;

break;

}

bool flag = true;

for (auto here : ret) {

if (now % here == 0) {

flag = false;

break;

}

}

if (flag) {

ret.push\_back(now);

}

}

}

ctn = true;

int ret\_sqrt\_cnt = (int)ret.size();

for (i = mid\_point - 2; ctn && i <= (N - 1) / 6 + 1; i++) {

for (j = -1; j <= 1; j += 2) {

int now = i \* 6 + j;

if (now <= ret[ret\_sqrt\_cnt - 1]) continue;

if (now > N) {

ctn = false;

break;

}

bool flag = true;

for (k = 0; k<ret\_sqrt\_cnt; k++) {

if (now % ret[k] == 0) {

flag = false;

break;

}

}

if (flag) {

ret.push\_back(now);

}

}

}

return ret;

}

//return <prime number, power\_cnt>

//ex) N = 12 / return vector<pair<2, 2>, pair<3, 1>>

vector<pair<int, int>> factorize(int N) {

auto primes = getPrimes(sqrt(N) + 5);

return factorize(N, primes);

}

vector<pair<int, int>> factorize(int N, vector<int> primes) {

vector<pair<int, int>> ret;

for (auto p : primes) {

int c = 0;

while (N % p == 0) {

N /= p;

c++;

}

if (c>0) ret.push\_back(make\_pair(p, c));

}

if (N > 1) ret.push\_back(make\_pair(N, 1));

return ret;

}

//extended gcd function

//returns gcd(a, b) by value,

//and x, y by reference that satisfies ax + by = gcd(a, b)

//Complexity : 12log2/(pi^2) log a + O(1) approximated by "0.85loga + O(1) in average case",

// "O(logb) in worst case" when a>=b

template <typename T>

T xGCD(T a, T b, T \*x, T \*y) {

if (a == 0) {

\*x = 0;

\*y = 1;

return b;

}

T x1, y1;

T gcd = xGCD(b%a, a, &x1, &y1);

\*x = y1 - (b / a) \* x1;

\*y = x1;

return gcd;

}

//m SHOULD BE PRIME NUMER!! It doesn't make any assertion!

//returns multiplicative inverse by modulo

//ex) mul\_inverse\_modulo(3, 11) = 4 since 3 \* 4 is equivalent with 1 by modulo 11

//Complexity : O( (log m)^2 )

template <typename T>

T mul\_inverse\_modulo(T a, T m) {

T x, y;

xGCD(a, m, &x, &y);

return x;

}

//returns ( n C r ) % MOD without caching in

template <typename T>

T combination(T n, T r, T MOD) {

if (r > n / 2) r = n - r;

T ret = 1;

for (T i = n; i >= n - r + 1; i--) {

ret \*= i;

ret %= MOD;

}

for (T i = r; i >= 1; i--) {

ret \*= mul\_inverse\_modulo(i, MOD);

ret %= MOD;

}

return ret;

}

template<typename T>

T gcd(T a, T b) {

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

}

// U must cover T

template<typename T, typename U>

U lcm(T a, T b) {

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

}

void Combination(int\*\* c, int n) {

int i, j;

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

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

c[i][j] = 0;

}

}

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

c[i][0] = 1;

}

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

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

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

//if there's a modulo opeartion remove comment mark and add new parameter MOD

//c[i][j] %= MOD;

}

}

}

//Created by Maybe 10/3/15

//chinese\_remainder\_Theorem

/\* if there is a possibility of k being very big, then prime factorize m[i],

\* find modular inverse of 'temp' of each of the factors

\* 'k' equals to the multiplication ( modular mods[i] ) of modular inverses

\*/

template <typename type>

type chinese\_remainder(const vector<type>& r, const vector<type>& mods)

{

type M = 1;

for (size\_t i = 0; i<size\_t(mods.size()); i++) M \*= mods[i];

vector< type > m, s;

for (size\_t i = 0; i<size\_t(mods.size()); i++) {

m.push\_back(M / mods[i]);

type temp = m[i] % mods[i];

type k = 0;

while (true) {

if ((k\*temp) % mods[i] == 1) break;

k++;

}

s.push\_back(k);

}

long long ret = 0;

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

ret += ((m[i] \* s[i]) % M \*r[i]) % M;

if (ret >= M) ret -= M;

}

return ret;

}

//Created by Maybe 10/3/15

//catalan\_number

/\*can refix binomial function to use lucas, pascal triangle for performance.

\*/

template<typename type>

long long int binomial(type n, type m)

{

if (n > m || n < 0) return 0;

long long int ans = 1, ans2 = 1;

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

ans \*= n - i;

ans2 \*= i + 1;

}

return ans/ans2;

}

template<typename type>

type catalan\_number(type n) {

return binomial(n \* 2, n) / (n + 1);

}

int main(void) {

return 0;

}

/\*PRIME NUMBERS

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,

101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199,

211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293,

307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397,

401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499,

503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599,

601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691,

701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,

809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887,

907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997

density of prime numbers : x / log x (lim x -> INF)

\*/

/\* PI

3.14159265358979323846264338327950288419716939937510582097494459230781640

\*/

// bitpartite\_matching.h

vector<vector<int> > v;

int backMatch[max\_N\*max\_N];

int column\_cnt[max\_N\*max\_N];

bool visited[max\_N\*max\_N];

bool dfs(int now) {

    if (visited[now]) return false;

    visited[now] = true;

for (auto next: v[now]) {

        if (backMatch[next] == -1 || dfs(backMatch[next])) {

            backMatch[next] = now;

            return true;

        }

    }

    return false;

}

int BitpartiteMatching() {

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

    int matched = 0;

for (size\_t i=0; i<v.size(); i++) {

        memset(visited, false, sizeof(visited));

        if (dfs(i)) matched++;

    }

    return matched;

}

// convex\_hull.h

template<typename type>

type cross(const pair<type,type> &O, const pair<type, type> &A, const pair<type, type> &B)

{

return (type)(A.first - O.first) \* (B.second - O.second) - (type)(A.second - O.second) \* (B.first - O.first);

}

template<typename type>

vector<pair<type,type> > convex\_hull(vector<pair<type, type> > map)

{

int k = 0;

vector<pair<type, type> > result(2 \* map.size());

sort(map.begin(), map.end(), [](pair<type,type> p, pair<type,type> q) { return p.second > q.second || ((!(p.second < q.second) && p.first < q.first)); });

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

{

while (k >= 2 && cross<type>(result[k - 2], result[k - 1], map[i]) <= 0)

k--;

result[k++] = map[i];

}

for (int i = map.size() - 2, t = k + 1; i >= 0; i--)

{

while (k >= t && cross<type>(result[k - 2], result[k - 1], map[i]) <= 0)

k--;

result[k++] = map[i];

}

result.resize(k);

return result;

}

// dijkstra.h

//map[i][j]={x,y}--i:index,j:start,x:end,y:cost

//dist[i]=j--i:end,j:cost

//start

template<typename t\_index, typename t\_cost>

void dijkstra(const vector<vector<pair<t\_index, t\_cost>>>& map, vector<t\_cost>& dist, const size\_t& start)

{

priority\_queue<pair<t\_index, t\_cost>> pq;

pq.push({ start, 0 });

while (pq.size())

{

pair<t\_index, t\_cost> now = pq.top();

pq.pop();

for (int i = 0; i < map[now.first].size(); i++)

{

t\_index next\_index = map[now.first][i].first;

if (dist[next\_index] > -now.second + map[now.first][i].second)

{

dist[next\_index] = -now.second + map[now.first][i].second;

pq.push({ next\_index, -dist[next\_index] });

}

}

}

}

// eulerCircuit.h

class euler {

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

public:

euler(vector<int> map)

{

this->V = map.size();

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

this->addEdge(i,map.at(i));

}

euler(int V) {this->V = V; adj = new list<int>[V]; }

~euler() { delete [] adj; }

void addEdge(int v, int w);

int isEulerian();

bool isConnected();

void DFSUtil(int v, bool visited[]);

};

void euler::addEdge(int v, int w)

{

adj[v].push\_back(w);

adj[w].push\_back(v); // Note: the graph is undirected

}

void euler::DFSUtil(int v, bool visited[])

{

visited[v] = true;

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFSUtil(\*i, visited);

}

bool euler::isConnected()

{

bool visited[V];

int i;

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

visited[i] = false;

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

if (adj[i].size() != 0)

break;

if (i == V)

return true;

DFSUtil(i, visited);

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

if (visited[i] == false && adj[i].size() > 0)

return false;

return true;

}

int euler::isEulerian()

{

// Check if all non-zero degree vertices are connected

if (isConnected() == false)

return 0;

// Count vertices with odd degree

int odd = 0;

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

if (adj[i].size() & 1)

odd++;

// If count is more than 2, then graph is not Eulerian

if (odd > 2)

return 0;

// If odd count is 2, then semi-eulerian.

// If odd count is 0, then eulerian

// Note that odd count can never be 1 for undirected graph

return (odd)? 1 : 2;

}

// gauss\_jordan.h

template <typename type>

type GaussJordan(vector<vector<type> > &a, vector<vector<type> > &b) {

const int n = a.size();

const int m = b[0].size();

vector<int> irow(n), icol(n), ipiv(n);

type det = 1;

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

int pj = -1, pk = -1;

for (int j = 0; j < n; j++) if (!ipiv[j])

for (int k = 0; k < n; k++) if (!ipiv[k])

if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }

if (fabs(a[pj][pk]) < 1e-10) { return -1; }

ipiv[pk]++;

swap(a[pj], a[pk]);

swap(b[pj], b[pk]);

if (pj != pk) det \*= -1;

irow[i] = pj;

icol[i] = pk;

type c = 1.0 / a[pk][pk];

det \*= a[pk][pk];

a[pk][pk] = 1.0;

for (int p = 0; p < n; p++) a[pk][p] \*= c;

for (int p = 0; p < m; p++) b[pk][p] \*= c;

for (int p = 0; p < n; p++) if (p != pk) {

c = a[p][pk];

a[p][pk] = 0;

for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] \* c;

for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] \* c;

}

}

for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {

for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);

}

return det;

}

// hamiltonCycle.h

//input vector<vector<boo>> map;

bool hamiltonCycle\_sub(vector<vector<bool> > map, vector<int> path, int pos)

{

if(pos == map.size())

if(map[path[pos-1]][path[0]]==1)

return true;

else

return false;

for(int i = 1; i < map.size(); i++)

{

if(map[path[pos-1]][i])

{

bool chk = true;

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

if(path[j]==i)

{

chk = false;

break;

}

if(chk)

{

path[pos] = i;

if(hamiltonCycle\_sub(map,path,pos+1))

return true;

path[pos] = -1;

}

}

}

return false;

}

bool hamiltonCycle(vector<vector<bool> > map)

{

vector<int> path(map.size(),-1);

path[0] = 0;

if(!hamiltonCycle\_sub(map,path,1))

{

//No solution

return false;

}

//path == solution

return true;

}

// suffixArray.h

// INPUT: string s

// OUTPUT: array suffix[] such that suffix[i] = index (from 0 to L-1)

// of substring s[i...L-1] in the list of sorted suffixes.

// That is, if we take the inverse of the permutation suffix[],

// we get the actual suffix array.

struct SuffixArray {

const int L;

string s;

vector<vector<int> > P;

vector<pair<pair<int,int>,int> > M;

SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L) {

for (int i = 0; i < L; i++) P[0][i] = int(s[i]);

for (int skip = 1, level = 1; skip < L; skip \*= 2, level++) {

P.push\_back(vector<int>(L, 0));

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

M[i] = make\_pair(make\_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);

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

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

P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;

}

}

vector<int> GetSuffixArray() { return P.back(); }

// returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]

int LongestCommonPrefix(int i, int j) {

int len = 0;

if (i == j) return L - i;

for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {

if (P[k][i] == P[k][j]) {

i += 1 << k;

j += 1 << k;

len += 1 << k;

}

}

return len;

}

};

// topologicalSort.h

//위상정렬. DAG가 아니면 빈 벡터 반환

template<typename type>

void topologicalSort\_sub(size\_t index, vector<type>& map, vector<type>& result, vector<type>& order)

{

result[index] = 1;

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

if(map[index][i] && !result[i])

topologicalSort\_sub(i,map,result,order);

order.push\_back(index);

}

template<typename type>

vector<type> topologicalSort(vector<type> map)

{

vector<type> order, result(map.size(),0);

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

if(!result[i])

topologicalSort\_sub(i,map,result,order);

reverse(order.begin(), order.end());

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

for(int j = i + 1; j < map.size(); j++)

if(map[order[j]][order[i]])

return vector<type>();

return order;

}