TIPS:

1. visita todas las componentes del grafo!

inverse(a) = (inverse(prime % a) \* (prime - prime/a)) % prime

#include <iostream>

#include <algorithm>

#include <stdio.h>

#include <string>

#include <vector>

#include <map>

#include <math.h>

#include <numeric>

#include <queue>

#include <stack>

#include <utility>

#include <queue>

#include <set>

#include <iomanip>

#include <sstream>

using namespace std;

typedef int64\_t ll;

typedef long double ld;

typedef pair<int,int> pii;

typedef vector<pii> vii;

typedef vector<int> vi;

#define MINF -1000000000000

#define MOD 1000000007

#define m(a,b) (a%b + b)%b

#define loop(i,a,b) for(int i = a; i < b; ++i)

#define invl(i,a,b) for(int i = a; i > b; --i)

const int maxN = 100005;

int main(){

ios\_base::sync\_with\_stdio(0); cin.tie(0);

if(fopen("case.txt", "r")) freopen("case.txt", "r", stdin);

}

vi adj[maxN];

int papa[maxN];

bool visited[maxN];

int level[maxN];

DFS

void dfs(int f, int nd){

for(int son : adj[nd]){

if(son == f) continue

//si eres back edge

if(visited[son] && level[son] < level[nd]){

}

//si eres front edje

else if(!visited[son]){

dfs(nd, son);

}

}

}

void recorridoInicial(int nd){

for(pill h: arbol[nd])

if(h.first != papa[nd])

{

papa[h.first] = nd;

++cantHijos[nd];

recorridoInicial(h.first);

}

}

Puentes

// =========== Puentes

/\*

dp[u] = numero de back edges que van de un "hijo" de u a un "ancestro" de u

puente entre a y b syss

dp[u] = 0

calculo de dp[u] = Suma(dp[hijos]) - backedgesQueLLeganAU + backEdgedSalenU

\*/

int dp[maxN];

int llegan[maxN];

bool hayPuente;

void dfs(int f, int nd){

visited[nd] = true;

level[nd] = level[f] + 1;

int salen = 0;

for(int son : adj[nd]){

if(son == f) continue;

//si eres back edge

if(visited[son] && level[son] < level[nd]){

++salen;

++llegan[son];

roads.push\_back(pii(nd, son));

}

//si eres front edje

else if(!visited[son]){

dfs(nd, son);

dp[nd] += dp[son];

roads.push\_back(pii(nd, son));

}

}

dp[nd] = dp[nd] - llegan[nd] + salen;

if(nd != 1 && dp[nd] == 0)

hayPuente = true;

}

loop(i, 0, m){

int a, b;

cin>>a>>b;

adj[a].push\_back(b);

adj[b].push\_back(a);

}

BFS

void bfs(int i){

queue<pii> c;

c.push(pii(i,0));

int nd, v;

fill(visited, visited + n + 5, 0);

while(!c.empty()){

nd = c.front().first; v = c.front().second;

c.pop();

if(!visited[nd]){

visited[nd] = true;

//meto a mis hijos con distancia mas uno

for(int h: adj[nd])

c.push(pii(h, v+1));

}

}

}

//=================== LCA

//asumiento que los nodos van de 1 a n

// si van de 0 a n-1 cambia el create LCA Table en el for nd = 0; nd < n

//siempre asigna a la raiz como su propio padre

// y su nivel como 1 o 0 dependiendo de que te convenga

//en main agrega

// assignParents(raiz, raiz);

// createLCATable();

struct node{

int level;

}nodes[maxN+1];

Const int LogN = 18;

int table[LogN+1][maxN+1], n;

void assignParents(int nd, int fa = nd){

for(int son: adj[nd]){

if(son != fa){

nodes[son].level = 1 + nodes[nd].level;

table[0][son] = nd;

//aqui puedes aprovechar para agregar funcionalidad que ocupes para el

//problema especifico

assignParents(son, nd);

}

}

}

//creates LCA Binary Lifting table

void createLCATable(){

int mid;

for(int k = 1; k<=LogN; ++k){

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

mid = table[k-1][nd];

table[k][nd] = table[k-1][mid];

}

}

}

//to get the k-th father of node i

int jump(int i, int k){

int x=0;

while(k > 0){

if(k&1)

i = table[x][i];

++x;

k>>=1;

}

return i;

}

int lca(int i, int j){

//to put i and j in the same level

if(nodes[i].level< nodes[j].level)

j = jump(j, nodes[j].level-nodes[i].level);

else

i = jump(i, nodes[i].level-nodes[j].level);

//these happens if one of them was lca of them

if(i==j)

return i;

//these is to put both just below the LCA

for(int d = LogN; d>=0; --d){

if(table[d][i] != table[d][j]){

i = table[d][i];

j = table[d][j];

}

}

//as they were just below the LCA, the LCA is the parent of either of the nodes

return table[0][i];

}

Bin search

int inf = 1, sup = r, mitad;

while(inf != sup){

mitad = (inf + sup)/2 +1;

if(sePuede(mitad))

inf = mitad;

else

sup = mitad - 1;

}

//=============== Exponenciacion bianria

ll binPow(ll base, ll exp){

ll r = 1;

while (exp > 0){

if(exp & 1)

r = (r\*base) % MOD;

exp >>= 1;

base = (base \* base) % MOD;

}

return r%MOD;

}

Fenwick Tree

void update(int pos, ll x){

while(pos <= l){

bit[pos] = bit[pos] + x;

pos += pos&(-pos);

}

}

ll query(int pos){

ll suma = 0;

while(pos > 0){

suma = suma + bit[pos];

pos -= pos&(-pos);

}

return suma;

}

ll query(int pos1, int pos2){

return query(pos2) - query(pos1-1);

}

ll binPow(ll base, ll exp){

ll r = 1;

while (exp > 0){

if( exp & 1)

r = (r\*base) % MOD;

exp >>= 1;

base = (base \* base) % MOD;

}

return r;

}

//gets the greatest common divisor of two numbers

int gcd(int a, int b){

if(a == 0)

return b;

else

return gcd(b%a, a);

}

/\*

convierte un string a un string con caracteres ordenados

y guarda en bucket la cantidad de veces que esta una letra en el string inicial

regresa el tamanio del string final

a - string inicial

bucket - vector que tiene cuantos hay de cada letra

s - string que guarda la lista de caracteres ordenados

ejemplo:

IN: aaaazzzbbb

Out: s = abz

bucket = {4, 3, 3}

regresa 3

\*/

int bucketString(string a, vi &bucket, string &s){

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

int sizeS = 1;

s = a.substr(0, 1);

bucket.push\_back(0);

for(char c : a){

if( c == s[sizeS-1])

++bucket[sizeS-1];

else

{

s += c;

bucket.push\_back(1);

++sizeS;

}

}

}

int binSearch(int m){

ll ini = 0, fin = 1e9, mitad;

while(ini != fin){

mitad = (ini+fin)/2;

if(true)

fin = mitad;

else

ini = mitad+1;

}

return ini;

}

Criba

const int maxN = 1e7;

vector <int> prime;

bool primo[maxN];

void sieve (int n) {

fill (primo, primo + n, true);

primo[0] = primo[1] = false;

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

if (primo[i]) prime.push\_back(i);

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

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

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

}

}

}

Eulers Totient function

vector <int> prime;

bool is\_composite[maxN];

int eulerTotient[maxN];

void sieve (int n) {

fill (is\_composite, is\_composite + n, false);

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

if (!is\_composite[i]){

prime.push\_back(i);

eulerTotient[i] = i-1;

}

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

is\_composite[i \* prime[j]] = true;

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

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

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

break;

}

}

}

}

Maximum Flow

Implementa ford Fulkerson con bfs

O(V\*(E^2))

**int n;**

**vector<vector<int>> capacity; //tamanio v\*v**

**// capacity[i][j] es lo que puede //ir de i a j, como es dirigido, el otro tiene cero**

**vector<vector<int>> adj; //adj es bidireccional**

**int bfs(int s, int t, vector<int>& parent) {**

**fill(parent.begin(), parent.end(), -1);**

**parent[s] = -2;**

**queue<pair<int, int>> q;**

**q.push({s, INF});**

**while (!q.empty()) {**

**int cur = q.front().first;**

**int flow = q.front().second;**

**q.pop();**

**for (int next : adj[cur]) {**

**if (parent[next] == -1 && capacity[cur][next]) {**

**parent[next] = cur;**

**int new\_flow = min(flow, capacity[cur][next]);**

**if (next == t)**

**return new\_flow;**

**q.push({next, new\_flow});**

**}**

**}**

**}**

**return 0;**

**}**

**int maxflow(int s, int t) {**

**int flow = 0;**

**vector<int> parent(n);**

**int new\_flow;**

**while (new\_flow = bfs(s, t, parent)) {**

**flow += new\_flow;**

**int cur = t;**

**while (cur != s) {**

**int prev = parent[cur];**

**capacity[prev][cur] -= new\_flow;**

**capacity[cur][prev] += new\_flow;**

**cur = prev;**

**}**

**}**

**return flow;**

**}**

Maximum Bipartite Matching

Todo se basa en:

El emparejamiento bipartito actual es máximo si y sólo si no hay un “augmenting path” en el grafo.

Un augmenting path es un camino que alterna entre aristas libres y ocupadas y empieza y termina con un nodo libre.

Algoritmo:

1) Initialize Maximal Matching M as empty.

2) While there exists an Augmenting Path p

Remove matching edges of p from M and add not-matching edges of p to M

(This increases size of M by 1 as p starts and ends with a free vertex)

3) Return M.

// ================ MAX BIPARTITE MATCHING

//instructions:

// create bipartite graph normally

// run dfs to determine the side of each vertex

// run maxBPM()

// global variable

// n total size of both sides

const int maxN = 505;

int n, finalNode;

int visited[maxN], side[maxN], match[maxN], parent[maxN];

vector<int> adj[maxN];

#define noOne maxN+1

#define LEFT 0

#define RIGHT 1

void dfs(int v, int parity){

visited[v] = true;

side[v] = parity&1? LEFT : RIGHT;

for(int son : adj[v]){

if(!visited[son])

dfs(son, parity^1);

}

}

bool BFS(){

queue<int> bfs;

bool newPath = false;

fill(visited, visited+n, 0);

//free vertex from left side into the bfs

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

if(match[i] == noOne && side[i] == LEFT){

bfs.push(i);

visited[i] = true;

}

}

//perform bfs

while(!bfs.empty()){

int v = bfs.front();

bfs.pop();

// we are on the right side looking for

// a matching edge

if(side[v] == RIGHT){

//if its a free vertex I've already finished

if(match[v] == noOne){

newPath = true;

finalNode = v;

break;

}

else{

int myMatch = match[v];

//I mark it as visited and add it to bfs

//I'm the only one who can visit it in bfs

visited[myMatch] = true;

parent[myMatch] = v;

bfs.push(myMatch);

}

}

// we are on the left side looking for

// a non-matching edge

else{

for(int son : adj[v]){

//if it's visited already, I don't visit it

// i don't check whether it's my match or not because

// only my match can push me to bfs and it's already visited

if(visited[son]) continue;

//I mark it as visited and add it to bfs

visited[son] = true;

parent[son] = v;

bfs.push(son);

}

}

}

return newPath;

}

int maxBPM(){

int sizeMatching = 0;

fill(match, match+n, noOne);

while(BFS()){

++sizeMatching;

int v = finalNode;

int parity = 1;

while(parent[v] != v){

//we reassign matching nodes in odd number steps

if(parity){

match[v] = parent[v];

match[parent[v]] = v;

}

parity &= 1;

v = parent[v];

}

}

return sizeMatching;

}

//=================== FIN MAXIMUM BIPARTITE MATCHING

int main(){

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

if(!visited[i])

dfs(i, 1);

cout<<maxBPM()<<endl;

return 0;

}

//=== precalcula factorial

ll fact[maxN], invFact[maxN];

ll binPow(ll base, ll exp){

ll r = 1;

while (exp > 0){

if( exp & 1)

r = (r\*base) % MOD;

exp >>= 1;

base = (base \* base) % MOD;

}

return r;

}

ll comb(int num, int k){

if (num < 0 || num < k) return 0;

return (((fact[num]\*invFact[k])%MOD)\*invFact[num-k])%MOD;

}

fact[0] = invFact[0] = 1;

for(int i = 1; i <= 2\*1e5 + 5; ++i){

fact[i] = (fact[i-1]\*i)%MOD;

invFact[i] = binPow(fact[i], MOD-2);

}

%string a int numero

int toInt(string numero){

int resp = 0;

for(char c : numero){

resp = 10\*resp + (c-'0');

}

return resp;

}