|  |  |
| --- | --- |
| Divide and conquer  Finding interesting points in N log N  Algorithm analysis  Master theorem  Amortized time complexity  Greedy algorithm  Scheduling  Kadane’s algorithm  Invariants  Huffman encoding  Graph theory  Dynamic graphs (book-keeping)  BFS/DFS/ 0-1 BFS  DFS tree  Dijkstra's algorithm  MST: Prim's algorithm  Bellman-Ford  \*\*\*Min-cost max flow  \*\*\*Matrix tree theorem  Floyd-Warshall  Euler cycles  Bridge  Bipartite matching  Topological sorting  Strongly connected components  2-SAT  Edge coloring  Trees  Vertex coloring  \* Bipartite graphs (=> trees)  Diameter and centroid  K'th shortest path  LCA  Shortest cycle  Dynamic programming  Knapsack  Coin change  LCS / LIS (segtree)  Number of paths in a dag  Shortest path in a dag  Dynprog over intervals, subsets, probabilities, trees  Divide and conquer  Knuth optimization  Convex hull optimizations  RMQ (sparse table a.k.a 2^k-jumps)  \*\*\*Log partitioning (loop over most restricted)  Combinatorics  Compute binomial coefficients  Pigeon-hole principle  Inclusion/exclusion  \*\*\*Catalan number  \*\*\*Pick's theorem  Combinatorial search  Meet in the middle  Brute-force with pruning  Bidirectional search  Sorting  Radix sort | Number theory  Integer parts  Divisibility  Euclidean algorithm  Modular arithmetic  \*\*\*Chinese remainder theorem  Fermat's little theorem  \*\*Euler's theorem  \*\*Phi function  \*\*\*Pollard-Rho  Game theory  Combinatorial games  Mini-max  Nim/ Grundy numbers  Games on trees /graphs  General/Bipartite games - repetition  Alpha-beta pruning  Probability theory  Optimization  Binary, Ternary search  Unimodality and convex functions  Binary search on derivative  Numerical methods  Numeric integration  \*\*Newton's method  Root-finding with binary/ternary search  Matrices  \*\*\*Gaussian elimination  Exponentiation by squaring  Geometry  Coordinates and vectors  \* Cross product  \* Scalar product  Convex hull  Polygon cut  Closest pair  Coordinate-compression  Quadtrees  Strings  Longest common substring  Palindrome subsequences  Knuth-Morris-Pratt  Tries  Rolling polynomial hashes  Suffix array / Suffix tree / Automaton  Aho-Corasick  Manacher's algorithm  Letter position lists  Data structures  LCA (2^k-jumps in trees in general)  Pull/push-technique on trees  Lazy propagation  Self-balancing trees  Convex hull trick  Monotonic queues / stacks / sliding queues  Persistent segment tree |

**Initial Template**

#include <bits/stdc++.h>  
using namespace std;

#pragma GCC optimize("trapv")

#define int long long  
using vi = vector<int>;  
using vb = vector<bool>;  
using vd = vector<double>;  
using vs = vector<string>;  
using pi = pair<int, int>;  
using vp = vector<pi>;  
using vvi = vector<vi>;  
using vvp = vector<vp>;  
using mi = map<int, int>;  
using si = set<int>;  
using msi = multiset<int>;

#define endl "\n"  
#define all(x) begin(x), end(x)  
#define F first  
#define S second  
#define PB(x) push\_back(x);  
#define MP make\_pair  
#define dbg(v) cout << #v << " = " << (v) << endl;  
auto nxt = [] { int x; cin >> x; return x;};  
const long long mod = 1000000007ll, mod2 = 998244353ll;

signed main() {  
 ios::*sync\_with\_stdio*(false);  
 cin.tie(nullptr);  
}

**Convex Hull Trick**

struct Line {

mutable ll k, m, p;

bool operator<(const Line& o) const { return k > o.k; } // Reverse comparison for min

bool operator<(ll x) const { return p < x; }

};

struct LineContainer : multiset<Line, less<>> {

// (for doubles, use inf = 1/.0, div(a,b) = a/b)

static const ll inf = LLONG\_MAX;

ll div(ll a, ll b) { // floored division

return a / b - ((a ^ b) < 0 && a % b);

}

bool isect(iterator x, iterator y) {

if (y == end()) return x->p = inf, 0;

if (x->k == y->k) x->p = x->m < y->m ? inf : -inf; // Retain line with smaller intercept

else x->p = div(y->m - x->m, x->k - y->k);

return x->p >= y->p;

}

void add(ll k, ll m) {

auto z = insert({k, m, 0}), y = z++, x = y;

while (isect(y, z)) z = erase(z);

if (x != begin() && isect(--x, y)) isect(x, y = erase(y));

while ((y = x) != begin() && (--x)->p >= y->p)

isect(x, erase(y));

}

ll query(ll x) {

auto l = \*lower\_bound(x);

return l.k \* x + l.m;

}

};

**Topological sort // Kahn's algorithm**

vector<int> topological\_sort(const vector<vector<int>> &adj){

int n = adj.size();

vector<int> indegree(n, 0), ans;

ans.reserve(n);

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

for(auto it: adj[i])

++indegree[it];

queue<int> q;

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

if(!indegree[i]) q.push(i);

while(!q.empty()){

int top = q.front(); q.pop();

for(auto it: adj[top]) {

--indegree[it];

if(!indegree[it]) q.push(it);

}

ans.push\_back(top);

}

if(ans.size() != n)

ans[0] = -1;

return ans;

}

**Math**

**Pollard Rho**

using ll = long long;

namespace PollardRho {

mt19937 rnd(chrono::steady\_clock::now().time\_since\_epoch().count());

const int P = 1e6 + 9;

ll seq[P];

int primes[P], spf[P];

inline ll add\_mod(ll x, ll y, ll m) {

return (x += y) < m ? x : x - m;

}

inline ll mul\_mod(ll x, ll y, ll m) {

ll res = \_\_int128(x) \* y % m;

return res;

// ll res = x \* y - (ll)((long double)x \* y / m + 0.5) \* m;

// return res < 0 ? res + m : res;

}

inline ll pow\_mod(ll x, ll n, ll m) {

ll res = 1 % m;

for (; n; n >>= 1) {

if (n & 1) res = mul\_mod(res, x, m);

x = mul\_mod(x, x, m);

}

return res;

}

// O(it \* (logn)^3), it = number of rounds performed

inline bool miller\_rabin(ll n) {

if (n <= 2 || (n & 1 ^ 1)) return (n == 2);

if (n < P) return spf[n] == n;

ll c, d, s = 0, r = n - 1;

for (; !(r & 1); r >>= 1, s++) {}

// each iteration is a round

for (int i = 0; primes[i] < n && primes[i] < 32; i++) {

c = pow\_mod(primes[i], r, n);

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

d = mul\_mod(c, c, n);

if (d == 1 && c != 1 && c != (n - 1)) return false;

c = d;

}

if (c != 1) return false;

}

return true;

}

void init() {

int cnt = 0;

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

if (!spf[i]) primes[cnt++] = spf[i] = i;

for (int j = 0, k; (k = i \* primes[j]) < P; j++) {

spf[k] = primes[j];

if (spf[i] == spf[k]) break;

}

}

}

// returns O(n^(1/4))

ll pollard\_rho(ll n) {

while (1) {

ll x = rnd() % n, y = x, c = rnd() % n, u = 1, v, t = 0;

ll \*px = seq, \*py = seq;

while (1) {

\*py++ = y = add\_mod(mul\_mod(y, y, n), c, n);

\*py++ = y = add\_mod(mul\_mod(y, y, n), c, n);

if ((x = \*px++) == y) break;

v = u;

u = mul\_mod(u, abs(y - x), n);

if (!u) return \_\_gcd(v, n);

if (++t == 32) {

t = 0;

if ((u = \_\_gcd(u, n)) > 1 && u < n) return u;

}

}

if (t && (u = \_\_gcd(u, n)) > 1 && u < n) return u;

}

}

vector<ll> factorize(ll n) {

if (n == 1) return vector <ll>();

if (miller\_rabin(n)) return vector<ll> {n};

vector <ll> v, w;

while (n > 1 && n < P) {

v.push\_back(spf[n]);

n /= spf[n];

}

if (n >= P) {

ll x = pollard\_rho(n);

v = factorize(x);

w = factorize(n / x);

v.insert(v.end(), w.begin(), w.end());

}

return v;

}

}

**Phi function**

const int LIM = 5000000;

int phi[LIM];

void calculatePhi() {

rep(i,0,LIM) phi[i] = i&1 ? i : i/2;

for (int i = 3; i < LIM; i += 2) if(phi[i] == i)

for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;

}

**Sieve**

int N = 1e5;

vector<bool> is\_prime(N + 1, true);

void initializeSieve(){

is\_prime[0] = is\_prime[1] = false;

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

if (is\_prime[i])

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

is\_prime[j] = false;

}

set<int> primeSet(){

set<int> ans;

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

if(is\_prime[i]) ans.insert(ans.end(), i);

return ans;

}

**Matrix multiplication**

vvi mat\_mul(vvi &a, vvi &b, int md = mod){

int n=a.size(), cols=a[0].size(), m = b[0].size();;

vvi c(n, vi(m, 0));

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

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

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

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

c[i][j] %= md;

}

}

}

return c;

}

**Binary exp, modInverse, factorial, NchooseK**

const long long mod = 1000000007ll, mod2 = 998244353ll;

long long binaryExp(long long n, long long pow, long long m=mod){

long long ans = 1;

while(pow>0) {

if (pow & 1)

ans = ans \* n % m;

n = n \* n % m;

pow >>= 1;

}

return ans;

}

long long modInverse(long long n, long long m=mod){

return binaryExp(n, m-2, m);

}

vector<int> fact(100001, 1);

void initiateFact(int m=mod){

for(int i=2; i<fact.size(); ++i){

fact[i] = fact[i-1]\*i%m;

}

}

long long nChoosek(int n, int k){

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

if(k==0 || k==n) return 1;

if(k==1 || k==n-1) return n;

return fact[n]\* modInverse(fact[k]\*fact[n-k]%mod2, mod2) %mod2;

}

**DSU**

int N=1e5;

int parent[N+1], sz[N+1];

void init(int n=N) {

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

parent[i] = i;

sz[i] = 1;

}

}

int find\_rep(int a) {

if(parent[a] == a) return a;

return parent[a] = find\_rep(parent[a]);

}

void merge(int a, int b) {

a = find\_rep(a), b= find\_rep(b);

if(a==b) return;

if(sz[a] < sz[b])

swap(a, b);

parent[b] = a;

sz[a] += sz[b];

}

**String**

**Manacher**

vector<int> manacher(const string& arg){

string s;

for (auto c: arg)

s += string("#") + c;

s = '$' + s + string("#^");

int n = s.size();

vector<int> ans(n);

int l=2, r=2;

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

long long temp = min(r-i, ans[l+ (r-i)]);

ans[i] = max(0ll, temp);

while(s[i-ans[i]] == s[i+ans[i]]) ++ans[i];

if(i+ans[i] > r)

l = i - ans[i], r = i + ans[i];

}

return {ans.begin()+2, ans.end()-2};

}

**Z-Function**

vector<int> z\_function(string s) {

int n = s.size();

vector<int> z(n);

int l = 0, r = 0;

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

if(i < r) {

z[i] = min(r - i, z[i - l]);

}

while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {

z[i]++;

}

if(i + z[i] > r) {

l = i;

r = i + z[i];

}

}

return z;

}

**Rabin-Karp**

vector<int> rabin\_karp(string const& s, string const& t) {

const int p = 31;

const int m = 1e9 + 9;

int S = s.size(), T = t.size();

vector<long long> p\_pow(max(S, T));

p\_pow[0] = 1;

for (int i = 1; i < (int)p\_pow.size(); i++)

p\_pow[i] = (p\_pow[i-1] \* p) % m;

vector<long long> h(T + 1, 0);

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

h[i+1] = (h[i] + (t[i] - 'a' + 1) \* p\_pow[i]) % m;

long long h\_s = 0;

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

h\_s = (h\_s + (s[i] - 'a' + 1) \* p\_pow[i]) % m;

vector<int> occurrences;

for (int i = 0; i + S - 1 < T; i++) {

long long cur\_h = (h[i+S] + m - h[i]) % m;

if (cur\_h == h\_s \* p\_pow[i] % m)

occurrences.push\_back(i);

}

return occurrences;

}

**KMP**

vector<int> prefix\_function(string s) {

int n = (int)s.length();

vector<int> pi(n);

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

int j = pi[i-1];

while (j > 0 && s[i] != s[j])

j = pi[j-1];

if (s[i] == s[j])

j++;

pi[i] = j;

}

return pi;

}

**Trie**

const int K = 26;

struct Vertex {

int next[K];

bool output = false;

Vertex() {

fill(begin(next), end(next), -1);

}

};

vector<Vertex> trie(1);

void add\_string(string const& s) {

int v = 0;

for (char ch : s) {

int c = ch - 'a';

if (trie[v].next[c] == -1) {

trie[v].next[c] = trie.size();

trie.emplace\_back();

}

v = trie[v].next[c];

}

trie[v].output = true;

}

**Range Queries**

**Fenwick Tree**

class FenwickTree{

vector<long long> tree;

public:

FenwickTree(vector<long long> &arr){

int n = arr.size();

tree = vector<long long>(n+1, 0);

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

tree[i] = arr[i-1];

for(int j = i-1; j > i-(i&-i); j -= j&-j)

tree[i] += tree[j];

}

}

// 1-indexed

long long query(int i){

long long ans = 0;

while(i){

ans += tree[i];

i -= i&-i;

}

return ans;

}

// 1-indexed inclusive

long long query(int l, int r){

return query(r) - query(l-1);

}

// 1-indexed

void update(int i, long long delta){

while(i < tree.size()) {

tree[i] += delta;

i += i & -i;

}

}

};

**2D Fenwick Tree**

class FenwickTree2D{

vector<vector<long long>> tree;

int n, m;

public:

FenwickTree2D(vector<vector<long long>> &arr){

n = arr.size();

m = arr[0].size();

tree = vector<vector<long long>>(n+1, vector<long long>(n+1, 0));

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

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

tree[i][j] = arr[i-1][j-1];

for(int temp = j-1; temp> j-(j&-j); temp -= temp&-temp)

tree[i][j] += tree[i][temp];

}

}

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

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

for(int temp = i-1; temp>i-(i&-i); temp -= temp&-temp)

tree[i][j] += tree[temp][j];

}

long long rowquery(int i, int j){

long long ans = 0;

while(j) {

ans += tree[i][j];

j -= j & -j;

}

return ans;

}

long long query(int i, int j){

long long ans=0;

int temp = i;

while(temp){

ans += rowquery(temp, j);

temp -= temp&-temp;

}

return ans;

}

long long query(int i, int j, int x, int y){

long long a, b, c, d;

a = query(x, y);

b = query(x, j-1);

c = query(i-1, y);

d = query(i-1, j-1);

return a -b -c +d;

}

void updaterow(int i, int j, long long v){

while(j<=m){

tree[i][j] += v;

j += j&-j;

}

}

void update(int i, int j, long long v){

while(i<=n){

updaterow(i, j, v);

i += i&-i;

}

}

void display(){

for(const auto& it: tree){

for(auto jt: it) cout << jt << "\t";

cout << endl;

}

}

};

**Segment Tree**

class SegmentTree {

vector<long long> tree;

int n;

public:

SegmentTree(vector<long long> &arr) {

n = arr.size();

n = (n&-n)==n?n: 1<<\_\_lg(n)+1;

tree = vector<long long>(2\*n, INT\_MAX);

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

tree[i+n] = arr[i];

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

tree[i] = min(tree[2\*i] , tree[2\*i+1]);

}

void update(int i, int val) {

i += n;

tree[i] = val;

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

tree[i] = min(tree[i<<1] , tree[(i<<1)+1]);

}

int query(int l, int r, int l1=0, int r1 = -1, int i=1) {

if(r1 == -1) r1 = n-1;

if(l1 > r || r1 < l) return INT\_MAX;

if(l1 >= l && r1 <= r) return tree[i];

return min(query(l, r, l1, l1+(r1-l1)/2, 2\*i) , query(l, r, l1+(r1-l1+1)/2, r1, 2\*i+1));

}

};

**RMQ**

class RMQ {

vector<vector<long long>> rmq;

public:

RMQ(vector<long long> &arr) {

int n = arr.size();

int m = \_\_lg(n)+0;

rmq = vector<vector<long long>>(m, vector<long long>(n));

rmq[-1] = arr;

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

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

rmq[i][j] = min(rmq[i-2][j], rmq[i-1][j+(1<<i-1)]);

}

}

long long query(int l, int r) {

return min(rmq[\_\_lg(r-l+0)][l], rmq[\_\_lg(r-l+1)][r-(1<<\_\_lg(r-l+1)) +1]);

}

};

**Poly area**

template <class T> int sgn(T x) { return (x > 0) - (x < 0); }

template<class T>

struct Point {

typedef Point P;

T x, y;

explicit Point(T x=0, T y=0) : x(x), y(y) {}

bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }

bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }

P operator+(P p) const { return P(x+p.x, y+p.y); }

P operator-(P p) const { return P(x-p.x, y-p.y); }

P operator\*(T d) const { return P(x\*d, y\*d); }

P operator/(T d) const { return P(x/d, y/d); }

T dot(P p) const { return x\*p.x + y\*p.y; }

T cross(P p) const { return x\*p.y - y\*p.x; }

T cross(P a, P b) const { return (a-\*this).cross(b-\*this); }

T dist2() const { return x\*x + y\*y; }

double dist() const { return sqrt((double)dist2()); }

// angle to x-axis in interval [-pi, pi]

double angle() const { return atan2(y, x); }

P unit() const { return \*this/dist(); } // makes dist()=1

P perp() const { return P(-y, x); } // rotates +90 degrees

P normal() const { return perp().unit(); }

// returns point rotated 'a' radians ccw around the origin

P rotate(double a) const {

return P(x\*cos(a)-y\*sin(a),x\*sin(a)+y\*cos(a)); }

friend ostream& operator<<(ostream& os, P p) {

return os << "(" << p.x << "," << p.y << ")"; }

};

template<class T>

T polygonArea2(vector<Point<T>>& v) {

T a = v.back().cross(v[0]);

rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);

return a;

}

**Lazy Seg Tree**

const int inf = 1e9;

struct Node {

Node \*l = 0, \*r = 0;

int lo, hi, mset = inf, madd = 0, val = -inf;

Node(int lo,int hi):lo(lo),hi(hi){} // Large interval of -inf

Node(vi& v, int lo, int hi) : lo(lo), hi(hi) {

if (lo + 1 < hi) {

int mid = lo + (hi - lo)/2;

l = new Node(v, lo, mid); r = new Node(v, mid, hi);

val = max(l->val, r->val);

}

else val = v[lo];

}

int query(int L, int R) {

if (R <= lo || hi <= L) return -inf;

if (L <= lo && hi <= R) return val;

push();

return max(l->query(L, R), r->query(L, R));

}

void set(int L, int R, int x) {

if (R <= lo || hi <= L) return;

if (L <= lo && hi <= R) mset = val = x, madd = 0;

else {

push(), l->set(L, R, x), r->set(L, R, x);

val = max(l->val, r->val);

}

}

void add(int L, int R, int x) {

if (R <= lo || hi <= L) return;

if (L <= lo && hi <= R) {

if (mset != inf) mset += x;

else madd += x;

val += x;

}

else {

push(), l->add(L, R, x), r->add(L, R, x);

val = max(l->val, r->val);

}

}

void push() {

if (!l) {

int mid = lo + (hi - lo)/2;

l = new Node(lo, mid); r = new Node(mid, hi);

}

if (mset != inf)

l->set(lo,hi,mset), r->set(lo,hi,mset), mset = inf;

else if (madd)

l->add(lo,hi,madd), r->add(lo,hi,madd), madd = 0;

}

};

**Graph**

**2 SAT**

struct TwoSatSolver {

int n\_vars;

int n\_vertices;

vector<vector<int>> adj, adj\_t;

vector<bool> used;

vector<int> order, comp;

vector<bool> assignment;

TwoSatSolver(int \_n\_vars) : n\_vars(\_n\_vars), n\_vertices(2 \* n\_vars), adj(n\_vertices), adj\_t(n\_vertices), used(n\_vertices), order(), comp(n\_vertices, -1), assignment(n\_vars) {

order.reserve(n\_vertices);

}

void dfs1(int v) {

used[v] = true;

for (int u : adj[v]) {

if (!used[u])

dfs1(u);

}

order.push\_back(v);

}

void dfs2(int v, int cl) {

comp[v] = cl;

for (int u : adj\_t[v]) {

if (comp[u] == -1)

dfs2(u, cl);

}

}

bool solve\_2SAT() {

order.clear();

used.assign(n\_vertices, false);

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

if (!used[i])

dfs1(i);

}

comp.assign(n\_vertices, -1);

for (int i = 0, j = 0; i < n\_vertices; ++i) {

int v = order[n\_vertices - i - 1];

if (comp[v] == -1)

dfs2(v, j++);

}

assignment.assign(n\_vars, false);

for (int i = 0; i < n\_vertices; i += 2) {

if (comp[i] == comp[i + 1])

return false;

assignment[i / 2] = comp[i] > comp[i + 1];

}

return true;

}

void add\_disjunction(int a, bool na, int b, bool nb) {

// na and nb signify whether a and b are to be negated

a = 2 \* a ^ na;

b = 2 \* b ^ nb;

int neg\_a = a ^ 1;

int neg\_b = b ^ 1;

adj[neg\_a].push\_back(b);

adj[neg\_b].push\_back(a);

adj\_t[b].push\_back(neg\_a);

adj\_t[a].push\_back(neg\_b);

}

static void example\_usage() {

TwoSatSolver solver(3); // a, b, c

solver.add\_disjunction(0, false, 1, true); // a v not b

solver.add\_disjunction(0, true, 1, true); // not a v not b

solver.add\_disjunction(1, false, 2, false); // b v c

solver.add\_disjunction(0, false, 0, false); // a v a

assert(solver.solve\_2SAT() == true);

auto expected = vector<bool>(True, False, True);

assert(solver.assignment == expected);

}

};

**Floyd-Warshall**

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

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

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

if (d[i][k] < INF && d[k][j] < INF)

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

}

}

}

**Bellman-Ford**

void solve()

{

vector<int> d(n, INF);

d[v] = 0;

vector<int> p(n, -1);

for (;;) {

bool any = false;

for (Edge e : edges)

if (d[e.a] < INF)

if (d[e.b] > d[e.a] + e.cost) {

d[e.b] = d[e.a] + e.cost;

p[e.b] = e.a;

any = true;

}

if (!any)

break;

}

if (d[t] == INF)

cout << "No path from " << v << " to " << t << ".";

else {

vector<int> path;

for (int cur = t; cur != -1; cur = p[cur])

path.push\_back(cur);

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

cout << "Path from " << v << " to " << t << ": ";

for (int u : path)

cout << u << ' ';

}

}

**Dijkstra**

const int INF = 1000000000;

vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {

int n = adj.size();

d.assign(n, INF);

p.assign(n, -1);

vector<bool> u(n, false);

d[s] = 0;

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

int v = -1;

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

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

v = j;

}

if (d[v] == INF)

break;

u[v] = true;

for (auto edge : adj[v]) {

int to = edge.first;

int len = edge.second;

if (d[v] + len < d[to]) {

d[to] = d[v] + len;

p[to] = v;

}}}}

**SCC**

vector<bool> visited; // keeps track of which vertices are already visited

// runs depth first search starting at vertex v.

// each visited vertex is appended to the output vector when dfs leaves it.

void dfs(int v, vector<vector<int>> const& adj, vector<int> &output) {

visited[v] = true;

for (auto u : adj[v])

if (!visited[u])

dfs(u, adj, output);

output.push\_back(v);

}

// input: adj -- adjacency list of G

// output: components -- the strongy connected components in G

// output: adj\_cond -- adjacency list of G^SCC (by root vertices)

void scc(vvi const& adj, vvi&components, vector<vector<int>> &adj\_cond) {

int n = adj.size();

components.clear(), adj\_cond.clear();

vector<int> order; // will be a sorted list of G's vertices by exit time

visited.assign(n, false);

// first series of depth first searches

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

if (!visited[i])

dfs(i, adj, order);

// create adjacency list of G^T

vector<vector<int>> adj\_rev(n);

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

for (int u : adj[v])

adj\_rev[u].push\_back(v);

visited.assign(n, false);

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

vector<int> roots(n, 0); // gives the root vertex of a vertex's SCC

// second series of depth first searches

for (auto v : order)

if (!visited[v]) {

std::vector<int> component;

dfs(v, adj\_rev, component);

components.push\_back(component);

int root = \*min\_element(begin(component), end(component));

for (auto u : component)

roots[u] = root;

}

// add edges to condensation graph

adj\_cond.assign(n, {});

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

for (auto u : adj[v])

if (roots[v] != roots[u])

adj\_cond[roots[v]].push\_back(roots[u]);

}

**Max-Flow, Min-Cut Edmonds-Karp**

int n;

vector<vector<int>> capacity, adj;

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;

}

**Hamiltonian Path**

bool Hamiltonian\_path(vector<vector<int> >& adj, int N){

int dp[N][(1 << N)];

memset(dp, 0, sizeof dp);

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

dp[i][(1 << i)] = true;

for (int i = 0; i < (1 << N); i++) {

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

if (i & (1 << j)) {

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

if (i & (1 << k)

&& adj[k][j]

&& j != k

&& dp[k][i ^ (1 << j)]) {

dp[j][i] = true;

break;

}}}}}

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

if (dp[i][(1 << N) - 1])

return true;

}

return false;

}

**Eulerian path**

int main() {

int n;

vector<vector<int>> g(n, vector<int>(n));

// reading the graph in the adjacency matrix

vector<int> deg(n);

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

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

deg[i] += g[i][j];

}

int first = 0;

while (first < n && !deg[first])

++first;

if (first == n) {

cout << -1;

return 0;

}

int v1 = -1, v2 = -1;

bool bad = false;

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

if (deg[i] & 1) {

if (v1 == -1)

v1 = i;

else if (v2 == -1)

v2 = i;

else

bad = true;

}

}

if (v1 != -1)

++g[v1][v2], ++g[v2][v1];

stack<int> st;

st.push(first);

vector<int> res;

while (!st.empty()) {

int v = st.top();

int i;

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

if (g[v][i])

break;

if (i == n) {

res.push\_back(v);

st.pop();

} else {

--g[v][i];

--g[i][v];

st.push(i);

}

}

if (v1 != -1) {

for (size\_t i = 0; i + 1 < res.size(); ++i) {

if ((res[i] == v1 && res[i + 1] == v2) ||

(res[i] == v2 && res[i + 1] == v1)) {

vector<int> res2;

for (size\_t j = i + 1; j < res.size(); ++j)

res2.push\_back(res[j]);

for (size\_t j = 1; j <= i; ++j)

res2.push\_back(res[j]);

res = res2;

break;

}

}

}

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

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

if (g[i][j])

bad = true;

}

}

if (bad) {

cout << -1;

} else {

for (int x : res)

cout << x << " ";

}

}

**Order statistic tree**

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

using namespace \_\_gnu\_pbds;

typedef tree<int, null\_type, less<int>, rb\_tree\_tag, tree\_order\_statistics\_node\_update> ordered\_set; order\_of\_key, find\_by\_order

**LCA**

vvi anc(n, vi(20, -2)); vi depth(n, 0);

auto dfs = [&](auto &&self, int node, int p=-1)->void{

for(int c: tree[node]) {

if(c==p) continue;

anc[c][0] = node;

for(int i=0; i<20 && anc[c][i] != -2; ++i)

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

depth[c] = depth[node] + 1;

self(self, c, node);

}

};

dfs(dfs, 0);

auto lift = [&](int x, int k) {

while(k) {

x = anc[x][\_\_lg(k&-k)];

k -= k&-k;

}

return x;

};

auto lca = [&](int a, int b) {

if(depth[a] > depth[b])

swap(a, b);

b = lift(b, depth[b] - depth[a]);

if(a!=b)

for(int i=19; i>=0; --i)

if(anc[a][i] != anc[b][i]) {

a = anc[a][i];

b = anc[b][i];

}

return a==b? a: anc[a][0];

};