Contents

1 Setting

1.1 vimrc

2 Math

2.1 Basic Arithmetic

```
typedef long long ll;
typedef unsigned long long ull;

// calculate lg2(a)
inline int lg2(ll a)
{
    return 63 - __builtin_clzll(a);
}

// calculate the number of 1-bits
inline int bitcount(ll a)
{
    return __builtin_popcountll(a);
}

// calculate ceil(a/b)
// |a|, |b| <= (2^63)-1 (does not dover -2^63)
ll ceildiv(ll a, ll b) {
    if (b < 0) return ceildiv(-a, -b);
    if (a < 0) return (-a) / b;
    return ((ull)a + (ull)b - 1ull) / b;
}

// calculate floor(a/b)
// |a|, |b| <= (2^63)-1 (does not cover -2^63)
ll floordiv(ll a, ll b) {
    if (b < 0) return floordiv(-a, -b);
    if (a >= 0) return a / b;
    return -(ll)(((ull)(-a) + b - 1) / b);
}

// calculate a*b % m
// x86-64 only
ll large_mod_mul(ll a, ll b, ll m)
{
    return ll((__int128)a*(__int128)b%m);
}

// calculate a*b % m
// |m| < 2^62, x86 available
// O(logb)
ll large_mod_mul(ll a, ll b, ll m)
{
    a %= m; b %= m; ll r = 0, v = a;
    while (b) {
        if (b&1) r = (r + v) % m;
        b >>= 1;
        v = (v << 1) % m;
    }
    return r;
}

// calculate n^k % m
ll modpow(ll n, ll k, ll m) {
    ll ret = 1;
```

```
    n %= m;
    while (k) {
        if (k & 1) ret = large_mod_mul(ret, n, m);
        n = large_mod_mul(n, n, m);
        k /= 2;
    }
    return ret;
}

// calculate gcd(a, b)
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

// find a pair (c, d) s.t. ac + bd = gcd(a, b)
pair<ll, ll> extended_gcd(ll a, ll b) {
    if (b == 0) return { 1, 0 };
    auto t = extended_gcd(b, a % b);
    return { t.second, t.first - t.second * (a / b) };
}

// find x in [0,m) s.t. ax === gcd(a, m) (mod m)
ll modinverse(ll a, ll m) {
    return (extended_gcd(a, m).first % m + m) % m;
}

// calculate modular inverse for 1 ~ n
void calc_range_modinv(int n, int mod, int ret[]) {
    ret[1] = 1;
    for (int i = 2; i <= n; ++i)
        ret[i] = (ll)(mod - mod/i) * ret[mod%i] % mod;
}
```

2.2 Kirchoff's Theorem

ê·žë í ì ì €í šë í žëŠ¬ì ê° ì ë¥Œ êµ¬í ë ì ëŠ¬.

ë¬Ží ¥ ê·žë í ì Laplacian matrix Lë¥Œ ë§ë ë €. ìŽê² ì (ì ì ì ì°ši ë ê° í ë ¬) - (ìži í ë ¬)ìŽë €. Lì ì í ê³ Œ ìŽì í ë ì© ì ê±°í ê² ì L'ë Œ í ì . ìŽë í /ìŽìŽë êŽ ê³ ì ë €. ê·žë í ì ì €í šë í žëŠ¬ì ê° ì ë $det(L')$ìŽë €.

2.3 Matrix Operations

```
const int MATSZ = 100;

inline bool is_zero(double a) { return fabs(a) < 1e-9; }

// out = A^(-1), returns det(A)
// A becomes invalid after call this
// O(n^3)
double inverse_and_det(int n, double A[][MATSZ], double out[][MATSZ]) {
    double det = 1;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) out[i][j] = 0;
        out[i][i] = 1;
    }
    for (int i = 0; i < n; i++) {
        if (is_zero(A[i][i])) {
            double maxv = 0;
            int maxid = -1;
            for (int j = i + 1; j < n; j++) {
                auto cur = fabs(A[j][i]);
                if (maxv < cur) {
                    maxv = cur;
                    maxid = j;
                }
            }
            if (maxid == -1 || is_zero(A[maxid][i])) return 0;
            for (int k = 0; k < n; k++) {
                A[i][k] += A[maxid][k];
                out[i][k] += out[maxid][k];
            }
        }
        det *= A[i][i];
        double coeff = 1.0 / A[i][i];
        for (int j = 0; j < n; j++) A[i][j] *= coeff;
        for (int j = 0; j < n; j++) out[i][j] *= coeff;
        for (int j = 0; j < n; j++) if (j != i) {
            double mp = A[j][i];
            for (int k = 0; k < n; k++) A[j][k] -= A[i][k] * mp;
            for (int k = 0; k < n; k++) out[j][k] -= out[i][k] * mp;
        }
    }
    return det;
}
```

3 Data Structure

3.1 Idx Tree

```
#include <stdio.h>
#include <algorithm>
#include <vector>
using namespace std;
//Max index Tree
typedef struct IT{
    int n;
    vector<int> mx;
    IT(const vector<int>& a){
        n=a.size();
        mx.resize(n*4);
        init(a,0,n-1,1);
    }

    int init(const vector<int>& a,int l,int r,int n){
```

```
        if(l==r)mx[n]=a[l];
        int m=(l+r)/2;
        return mx[n]=max(init(a,l,m,n*2),init(a,m+1,r,n*2+1));
    }
    int quary(int l,int r,int n,int nl,int nr){
        if(r<nl||nr<l)return 0;
        if(l<=nl&&nr<r)return mx[n];
        int m=(l+r)/2;
        return max(quary(l,r,n*2,nl,m),quary(l,r,n*2+1,m+1,nr));
    }

    int quary(int l,int r){
        return quary(l,r,1,0,n-1);
    }

    int update(int i,int x,int n,int nl,int nr){
        if(i<nl&&nr<i)return mx[n];
        if(nl==nr)return mx[n]=x;
        int m = (l+r)/2;
        return mx[n]=max(update(i,x,n*2,nl,m), update(i,x,n*2+1,m+1,nr));
    }

    int update(int i, int x){
        return update(i,x,1,0,n-1);
    }
}IT;
```

3.2 Fenwick Tree

```
const int TSIZE = 100000;
int tree[TSIZE + 1];

// Returns the sum from index 1 to p, inclusive
int query(int p) {
    int ret = 0;
    for (; p > 0; p -= p & -p) ret += tree[p];
    return ret;
}

// Adds val to element with index pos
void add(int p, int val) {
    for (; p <= TSIZE; p += p & -p) tree[p] += val;
}
```

4 Graph

4.1 Ford Fulkerson

```
#include <stdio.h>
#include <vector>
#include <algorithm>
using namespace std;

typedef struct edge{
    int to;
    int cost;
    int from;
}edge;

vector<edge> v[11111];
int used[11111];
int flow(int s,int d,int mn){
    if(s==d)return mn;
    used[s]=1;
    for(int i=0;i<v[s].size();i++){
        edge e=v[s][i];
        if(used[e.to]==0&&e.cost>0){
            int x=flow(e.to,d,min(mn,e.cost));
            if(x>0){
                v[s][i].cost-=x;
                v[e.to][e.from].cost+=x;
                return x;
            }
        }
    }
    return 0;
}

// small n
vecetor<int> v[5010];
int flow[5010][5010];
int ford(int s,int d,int mn){
    if(s==d)return mn;
    used[s]=1;
    for(int i=0;i<v[s].size();i++){
        int e=v[s][i];
        if(used[e]==0&&flow[s][e]>0){
            int x=flow(e,d,min(mn,flow[s][e]));
            if(x>0){
                v[s][e]-=x;
                v[e][s]+=x;
                return x;
            }
        }
    }
    return 0;
}
```

4.2 Edmond Karp

```
const int INF = 987654321;
int n;
int cap[n][n], flow[n][n];

int edmond(int s, int t){
    memset(flow, 0, sizeof(flow));
    int totalflow = 0;
```

```
    while(true){
        vector<int> parent(n,-1);
        queue<int> q;
        parent[s]=s;
        q.push(s);
        while(!q.empty()){
            int here = q.front();q.pop();
            for(int there = 0;there < n;there++){
                if(cap[here][here] - flow[here][there] > 0 && parent[there] ==
                   -1){
                    q.push(there);
                    parent[there] = here;
                }
            }
        }
        if(parent[t] == -1)break;
        int amount = INF;
        for(int p=t;p!=s;p=parent[p]){
            amount = min(cap[parent[p]][p] - flow[parent[p]][p], amount);
        }
        for(int p=t;p!=s;p=parent[p]){
            flow[parent[p]][p]+=amount;
            flow[p][parent[p]]-=amount;
        }
        totalflow+=amount;
    }
    return totalflow;
}
```

4.3 Dijkstra

```
#include <stdio.h>
#include <algorithm>
#include <vector>
#include <queue>
using namespace std;
typedef pair<int,int> pii;
const int INF = 2147483647;

int n,m,a,b,c;
int d[1111];
vector<pii> g[1111];

void dijk(vector<pii>* g,int *d,int n,int k){ // start from k
    for(int i=0;i<n;i++){
        d[i]=INF;
    }
    d[k]=0;
    priority_queue<pii,vector<pii>,greater<pii> > pq;
    pq.push(pii(0,k));
    while(!pq.empty()){
        pii p=pq.top();pq.pop();
        int dis=p.first, v=p.second;
        if(d[v]<dis)continue;
        for(int i=0;i<g[v].size();i++){
            pii q=g[v][i];
            int to = q.first, cost = q.second;
            if(d[to] > cost+dis){
                pq.push(pii(d[to] = cost+dis,to));
            }
        }
    }
}
```

4.4 BCC, Cut vertex, Bridge

```
const int MAXN = 100;
vector<pair<int, int>> graph[MAXN];  // { next vertex id, edge id }
int up[MAXN], visit[MAXN], vtime;
vector<pair<int, int>> stk;

int is_cut[MAXN];            // v is cut vertex if  is_cut[v] > 0
vector<int> bridge;          // list of edge ids
vector<int> bcc_idx[MAXN];  // list of bccids for vertex i
int bcc_cnt;

void dfs(int nod, int par_edge) {
    up[nod] = visit[nod] = ++vtime;
    int child = 0;
    for (const auto& e : graph[nod]) {
        int next = e.first, edge_id = e.second;
        if (edge_id == par_edge) continue;
        if (visit[next] == 0) {
            stk.push_back({ nod, next });
            ++child;
            dfs(next, edge_id);
            if (up[next] == visit[next]) bridge.push_back(edge_id);
            if (up[next] >= visit[nod]) {
                ++bcc_cnt;
                do {
                    auto last = stk.back();
                    stk.pop_back();
                    bcc_idx[last.second].push_back(bcc_cnt);
                    if (last == pair<int, int>{ nod, next }) break;
                } while (!stk.empty());
                bcc_idx[nod].push_back(bcc_cnt);
                is_cut[nod]++;
            }
            up[nod] = min(up[nod], up[next]);
        }
        else
            up[nod] = min(up[nod], visit[next]);
    }
    if (par_edge == -1 && is_cut[nod] == 1)
        is_cut[nod] = 0;
}

// find BCCs & cut vertexs & bridges in undirected graph
// O(V+E)
void get_bcc() {
```

```
    vtime = 0;
    memset(visit, 0, sizeof(visit));
    memset(is_cut, 0, sizeof(is_cut));
    bridge.clear();
    for (int i = 0; i < n; ++i) bcc_idx[i].clear();
    bcc_cnt = 0;
    for (int i = 0; i < n; ++i) {
        if (visit[i] == 0)
            dfs(i, -1);
    }
}
```

4.5 Shortest Path Faster Algorithm

```
// shortest path faster algorithm
// average for random graph : O(E) , worst : O(VE)

const int MAXN = 20001;
const int INF = 100000000;
int n, m;
vector<pair<int, int>> graph[MAXN];
bool inqueue[MAXN];
int dist[MAXN];

void spfa(int st) {
    for (int i = 0; i < n; ++i) {
        dist[i] = INF;
    }
    dist[st] = 0;

    queue<int> q;
    q.push(st);
    inqueue[st] = true;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inqueue[u] = false;
        for (auto& e : graph[u]) {
            if (dist[u] + e.second < dist[e.first]) {
                dist[e.first] = dist[u] + e.second;
                if (!inqueue[e.first]) {
                    q.push(e.first);
                    inqueue[e.first] = true;
                }
            }
        }
    }
}
```

4.6 Bipartite Matching (Hopcroft-Karp)

```
// in: n, m, graph
// out: match, matched
// vertex cover: (reached[0][left_node] == 0) || (reached[1][right_node] == 1)
// O(E*sqrt(V))
struct BipartiteMatching {
    int n, m;
    vector<vector<int>> graph;
    vector<int> matched, match, edgeview, level;
    vector<int> reached[2];
    BipartiteMatching(int n, int m) : n(n), m(m), graph(n), matched(m, -1),
      match(n, -1) {}

    bool assignLevel() {
        bool reachable = false;
        level.assign(n, -1);
        reached[0].assign(n, 0);
        reached[1].assign(m, 0);
        queue<int> q;
        for (int i = 0; i < n; i++) {
            if (match[i] == -1) {
                level[i] = 0;
                reached[0][i] = 1;
                q.push(i);
            }
        }
        while (!q.empty()) {
            auto cur = q.front(); q.pop();
            for (auto adj : graph[cur]) {
                reached[1][adj] = 1;
                auto next = matched[adj];
                if (next == -1) {
                    reachable = true;
                }
                else if (level[next] == -1) {
                    level[next] = level[cur] + 1;
                    reached[0][next] = 1;
                    q.push(next);
                }
            }
        }
        return reachable;
    }

    int findpath(int nod) {
        for (int &i = edgeview[nod]; i < graph[nod].size(); i++) {
            int adj = graph[nod][i];
            int next = matched[adj];
            if (next >= 0 && level[next] != level[nod] + 1) continue;
            if (next == -1 || findpath(next)) {
                match[nod] = adj;
                matched[adj] = nod;
                return 1;
            }
        }
        return 0;
    }

    int solve() {
```

```
        int ans = 0;
        while (assignLevel()) {
            edgeview.assign(n, 0);
            for (int i = 0; i < n; i++)
                if (match[i] == -1)
                    ans += findpath(i);
        }
        return ans;
    }
};
```

4.7 Maximum Flow (Dinic)

```
// usage:
// MaxFlowDinic::init(n);
// MaxFlowDinic::add_edge(0, 1, 100, 100); // for bidirectional edge
// MaxFlowDinic::add_edge(1, 2, 100); // directional edge
// result = MaxFlowDinic::solve(0, 2); // source -> sink
// graph[i][edgeIndex].res -> residual
//
// in order to find out the minimum cut, use `l'.
// if l[i] == 0, i is unrechable.
//
// O(V*V*E)
// with unit capacities, O(min(V^(2/3), E^(1/2)) * E)
struct MaxFlowDinic {
    typedef int flow_t;
    struct Edge {
        int next;
        int inv; /* inverse edge index */
        flow_t res; /* residual */
    };
    int n;
    vector<vector<Edge>> graph;
    vector<int> q, l, start;

    void init(int _n) {
        n = _n;
        graph.resize(n);
        for (int i = 0; i < n; i++) graph[i].clear();
    }
    void add_edge(int s, int e, flow_t cap, flow_t caprev = 0) {
        Edge forward{ e, graph[e].size(), cap };
        Edge reverse{ s, graph[s].size(), caprev };
        graph[s].push_back(forward);
        graph[e].push_back(reverse);
    }
    bool assign_level(int source, int sink) {
        int t = 0;
        memset(&l[0], 0, sizeof(l[0]) * l.size());
        l[source] = 1;
        q[t++] = source;
        for (int h = 0; h < t && !l[sink]; h++) {
            int cur = q[h];
            for (const auto& e : graph[cur]) {
                if (l[e.next] || e.res == 0) continue;
                l[e.next] = l[cur] + 1;
                q[t++] = e.next;
            }
        }
        return l[sink] != 0;
    }
    flow_t block_flow(int cur, int sink, flow_t current) {
        if (cur == sink) return current;
        for (int& i = start[cur]; i < graph[cur].size(); i++) {
            auto& e = graph[cur][i];
            if (e.res == 0 || l[e.next] != l[cur] + 1) continue;
            if (flow_t res = block_flow(e.next, sink, min(e.res, current))) {
                e.res -= res;
                graph[e.next][e.inv].res += res;
                return res;
            }
        }
        return 0;
    }
    flow_t solve(int source, int sink) {
        q.resize(n);
        l.resize(n);
        start.resize(n);
        flow_t ans = 0;
        while (assign_level(source, sink)) {
            memset(&start[0], 0, sizeof(start[0]) * n);
            while (flow_t flow = block_flow(source, sink, numeric_limits<
              flow_t>::max()))
                ans += flow;
        }
        return ans;
    }
};
```

4.8 Min-cost Maximum Flow

```
#include <functional>
#include <queue>
#include <limits>
#include <vector>
#include <algorithm>

using namespace std;
// from KCM1700/algorithms

// precondition: there is no negative cycle.
// usage:
//  MinCostFlow mcf(n);
//  for(each edges) mcf.addEdge(from, to, cost, capacity);
//  mcf.solve(source, sink); // min cost max flow
//  mcf.solve(source, sink, 0); // min cost flow
//  mcf.solve(source, sink, goal_flow); // min cost flow with total_flow >=
  goal_flow if possible
struct MinCostFlow
```

```
{
    typedef int cap_t;
    typedef int cost_t;

    bool iszerocap(cap_t cap) { return cap == 0; }

    struct edge {
        int target;
        cost_t cost;
        cap_t residual_capacity;
        cap_t orig_capacity;
        size_t revid;
    };

    int n;
    vector<vector<edge>> graph;
    vector<cost_t> pi;
    bool needNormalize, ranbefore;
    int lastStart;

    MinCostFlow(int n) : graph(n), n(n), pi(n, 0), needNormalize(false),
      ranbefore(false) {}
    void addEdge(int s, int e, cost_t cost, cap_t cap)
    {
        if (s == e) return;
        edge forward={e, cost, cap, cap, graph[e].size()};
        edge backward={s, -cost, 0, 0, graph[s].size()};
        if (cost < 0 || ranbefore) needNormalize = true;
        graph[s].emplace_back(forward);
        graph[e].emplace_back(backward);
    }
    bool normalize(int s) {
        auto infinite_cost = numeric_limits<cost_t>::max();
        vector<cost_t> dist(n, infinite_cost);
        dist[s] = 0;
        queue<int> q;
        vector<int> v(n), relax_count(n);
        v[s] = 1; q.push(s);
        while(!q.empty()) {
            int cur = q.front();
            v[cur] = 0; q.pop();
            if (++relax_count[cur] >= n) return false;
            for (const auto &e : graph[cur]) {
                if (iszerocap(e.residual_capacity)) continue;
                auto next = e.target;
                auto ncost = dist[cur] + e.cost;
                if (dist[next] > ncost) {
                    dist[next] = ncost;
                    if (v[next]) continue;
                    v[next] = 1; q.push(next);
                }
            }
        }
        for (int i = 0; i < n; i++) pi[i] = dist[i];
        return true;
    }

    pair<cost_t, cap_t> AugmentShortest(int s, int e, cap_t flow_limit) {
        auto infinite_cost = numeric_limits<cost_t>::max();
        auto infinite_flow = numeric_limits<cap_t>::max();
        typedef pair<cost_t, int> pq_t;
        priority_queue<pq_t, vector<pq_t>, greater<pq_t>> pq;
        vector<pair<cost_t, cap_t>> dist(n, make_pair(infinite_cost, 0));
        vector<int> from(n, -1), v(n);

        if (needNormalize || (ranbefore && lastStart != s))
            normalize(s);
        ranbefore = true;
        lastStart = s;

        dist[s] = pair<cost_t, cap_t>(0, infinite_flow);
        pq.emplace(dist[s].first, s);
        while(!pq.empty()) {
            auto cur = pq.top().second; pq.pop();
            if (v[cur]) continue;
            v[cur] = 1;
            if (cur == e) continue;
            for (const auto &e : graph[cur]) {
                auto next = e.target;
                if (v[next]) continue;
                if (iszerocap(e.residual_capacity)) continue;
                auto ncost = dist[cur].first + e.cost - pi[next] + pi[cur];
                auto nflow = min(dist[cur].second, e.residual_capacity);
                if (dist[next].first <= ncost) continue;
                dist[next] = make_pair(ncost, nflow);
                from[next] = e.revid;
                pq.emplace(dist[next].first, next);
            }
        }
        /** augment the shortest path **/
        auto p = e;
        auto pathcost = dist[p].first + pi[p] - pi[s];
        auto flow = dist[p].second;
        if (iszerocap(flow)|| (flow_limit <= 0 && pathcost >= 0)) return pair<
          cost_t, cap_t>(0, 0);
        if (flow_limit > 0) flow = min(flow, flow_limit);
        /* update potential */
        for (int i = 0; i < n; i++) {
            if (iszerocap(dist[i].second)) continue;
            pi[i] += dist[i].first;
        }
        while (from[p] != -1) {
            auto nedge = from[p];
            auto np = graph[p][nedge].target;
            auto fedge = graph[p][nedge].revid;
            graph[p][nedge].residual_capacity += flow;
            graph[np][fedge].residual_capacity -= flow;
            p = np;
        }
        return make_pair(pathcost * flow, flow);
```

```
    }

    pair<cost_t,cap_t> solve(int s, int e, cap_t flow_minimum = numeric_limits
      <cap_t>::max()) {
        cost_t total_cost = 0;
        cap_t total_flow = 0;
        for(;;) {
            auto res = AugmentShortest(s, e, flow_minimum - total_flow);
            if (res.second <= 0) break;
            total_cost += res.first;
            total_flow += res.second;
        }
        return make_pair(total_cost, total_flow);
    }
};
```

5 Geometry

5.1 Operations

```
#include <stdio.h>

typedef struct point{
    int x,y;
}point;

int ccw(const point &a, const point &b, const point &c){
    return a.x*b.y+b.x*c.y+c.x*a.y - a.y*b.x-b.y*c.x-c.y*a.x;
}
```

6 String

6.1 KMP

```
#include <stdio.h>
#include <string.h>

void kmp(char *t, char *p, int *r, int *ff){
    int l = strlen(t)
    for(int i=0,j=0;i<l;i++){
        if(t[i]!=p[j]){
            if(j==0)r[i]=0;
            else{
                i--;
                j=ff[j-1];
            }
        }
        else r[i]=++j;
    }
}
int main(){
    int n;
    char a[1000], b[1000]; // a: 찾을문자열 , b: 대상문자열.
    int ff[1000], d[1000]; // ff: 실패함수배열 , d: 결과배열 .입력
    //
    kmp(a+1,a,ff+1,ff); // 실패함수생성
    kmp(b,a,d,ff);

}
```

7 etc..

7.1 Fast I/O

```
namespace fio {
    const int BSIZE = 524288;
    char buffer[BSIZE];
    int p = BSIZE;
    inline char readChar() {
        if(p == BSIZE) {
            fread(buffer, 1, BSIZE, stdin);
            p = 0;
        }
        return buffer[p++];
    }
    int readInt() {
        char c = readChar();
        while ((c < '0' || c > '9') && c != '-') {
            c = readChar();
        }
        int ret = 0; bool neg = c == '-';
        if (neg) c = readChar();
        while (c >= '0' && c <= '9') {
            ret = ret * 10 + c - '0';
            c = readChar();
        }
        return neg ? -ret : ret;
    }
}
```

7.2 Magic Numbers

ì ì : 10 007 , 10 009 , 10 111 , 31 567 , 70 001 , 1 000 003 , 1 000 033 , 4 000 037 , 1 000 000 007 , 1 000 000 009