TCR
October 29, 2016
git diff solution (Jens Heuseveldt, Ludo Pulles, Peter Ypma)

```
#include<bits/stdc++.h>

#define x first
#define y second

using namespace std;

typedef long long ll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef vector<int> vi;

const int INF = 2147483647; // (1 << 30) - 1 + (1 << 30)
const ll LLINF = 9223372036854775807LL; // (1LL << 62) - 1 + (1LL << 62)
const double pi = acos(-1.0);

// lambda-expression: [] (args) -> retType { body }

void run() {}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);
    // cerr << boolalpha;
    (cout << fixed).precision(10);
    int ntc;
    cin >> ntc;
    while (ntc--) { run(); }
    return 0;
}
```

Two prime numbers: *982451653*, *81253449*

0.1 Covering problems

$$\textit{Minimum edge cover} \iff \textit{Maximum independent set}$$

Matching
A set of edges without common vertices *(Maximum is the* ***largest*** *such set, maximal is a set which you cannot add more edges to without breaking the property).*

Minimum Vertex Cover
A set vertices (cover) such that each edge in the graph is incident to at least one vertex of the set.

Minimum Edge Cover
A set of edges (cover) such that every vertex is incident to at least one edge of the set.

Maximum Independent Set
A set of vertices in a graph such that no two of them are adjacent.

König's theorem
In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover

1 Mathematics

```

int abs(int x) { return x > 0 ? x : -x; }
int sign(int x) { return (x > 0) - (x < 0); }
```

```
// greatest common divisor
ll gcd(ll a, ll b) {
    while (b) {
        ll c = a % b;
        a = b; b = c;
    }
    return a;
}

// least common multiple
ll lcm(ll a, ll b) { return a / gcd(a, b) * b; }

// ax + by = gcd(a, b)
ll egcd(ll a, ll b, ll &x, ll &y) {
    ll r = b, rr = a, s = 0, ss = 1, t = 1, tt = 0, tmp;
    while (r) {
        ll q = rr / r;
        tmp = r; r = rr - q * r; rr = tmp;
        tmp = s; s = ss - q * s; ss = tmp;
        tmp = t; t = tt - q * t; tt = tmp;
    }
    x = ss; y = tt;
    return rr; // gcd
}

ll mod(ll a, ll m) {
    a %= m;
    return (a < 0) ? a + m : a;
}

const pll INVALID_CRT(0, -1);

// x = a0 mod b0 = a1 mod b1
pll crt(ll a1, ll m1, ll a2, ll m2) {
    ll x, y, gcd = egcd(m1, m2, x, y);
    if (a1 % gcd != a2 % gcd) return INVALID_CRT;
    return pll(mod(x * a2 * m1 + y * a1 * m2, m1 * m2) / gcd, m1 / gcd * m2);
}

// totient function for values 1 to N
int phi[N];

void sievePhi() {
    for (int i = N; i--; ) phi[i] = i;
    for (int i = 2; i <= N; i++)
        if (phi[i] == i)
            for (int j = i; j <= N; j += i)
                phi[j] -= phi[j] / i * (i - 1);
}

// modular exponentation: r = b^e mod m
ll modpow(ll b, ll e, ll m) {
    ll r = 1;
    while (e) {
        if (e & 1) r = (r * b) % m;
        e >>= 1;
        b = (b * b) % m;
    }
    return r;
}
```

2 Datastructures

2.1 Segment tree $\mathcal{O}(\log n)$

```
typedef /* Tree element */ S;
const int n = 1 << 20;
S t[2 * n];

// sum segment tree
S combine(S l, S r) { return l + r; }
// max segment tree
S combine(S l, S r) { return max(l, r); }

void build() {
    for (int i = n; --i > 0; )
        t[i] = combine(t[2 * i], t[2 * i + 1]);
}

// set value v on position p
void update(int p, int v) {
    for (t[p += n] = v; p /= 2; )
        t[p] = combine(t[2 * p], t[2 * p + 1]);
}

// sum on interval [l, r)
S query(int l, int r) {
    S resL, resR;
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
        if (l & 1) resL = combine(resL, t[l++]);
        if (r & 1) resR = combine(t[--r], resR);
    }
    return combine(resL, resR);
}
```

2.2 Binary Indexed Tree $\mathcal{O}(\log n)$

```
int bit[MAXN];

// arr[idx] += val
void update(int idx, int val) {
    while (idx < MAXN) bit[idx] += val, idx += idx & -idx;
}

// returns sum of arr[i], where i: [1, idx]
int query(int idx) {
    int ret = 0;
    while (idx) ret += bit[idx], idx -= idx & -idx;
    return ret;
}
```

2.3 Trie

```
struct trie {
    bool word;
    trie **child;

    trie() : word(false), child() {
        child = new trie*[26];
        for (int i = 26; i--; ) child[i] = NULL;
    }

    void addWord(const string &str)
    {
```

```
        trie *cur = this;
        for (char ch : str) {
            int idx = ch - 'a';
            if (cur->child[idx] == NULL)
                cur->child[idx] = new trie();
            cur = cur->child[idx];
        }
        cur->word = true;
    }

    bool isWord(const string &str)
    {
        trie *cur = this;
        for (char ch : str) {
            int idx = ch - 'a';
            if (cur->child[idx] == NULL) return false;
            cur = cur->child[idx];
        }
        return cur->word;
    }
};
```

2.4 Disjoint-Set / Union-Find $\mathcal{O}(\alpha(n))$

```
int par[MAXN], rnk[MAXN];

void uf_init(int n) {
    fill_n(par, n, -1);
    fill_n(rnk, n, 0);
}

int uf_find(int v) {
    return par[v] < 0 ? v : par[v] = uf_find(par[v]);
}

void uf_union(int a, int b) {
    if ((a = uf_find(a)) == (b = uf_find(b))) return;
    if (rnk[a] < rnk[b]) swap(a, b);
    if (rnk[a] == rnk[b]) rnk[a]++;
    par[b] = a;
}
```

3 Graph Algorithms

3.1 Maximum matching $\mathcal{O}(nm)$

This problem could be solved with a flow algorithm like Dinic's algorithm which runs in $\mathcal{O}(\sqrt{V}E)$, too.

```
bool vis[nodesRight]; // vis[rightnodes]
int par[nodesRight]; // par[rightnode] = leftnode
vector<int> adj[nodesLeft]; // adj[leftnode][i] = rightnode

bool match(int cur) {
    for (int nxt : adj[cur]) {
        if (vis[nxt]) continue;
        vis[nxt] = true;
        if (par[nxt] == -1 || match(par[nxt])) {
            par[nxt] = cur;
            return true;
        }
    }
    return false;
}
```

```

int matches = 0;
for (int i = 0; i < nodesLeft; i++) {
    memset(vis, false, sizeof(vis));
    if (match(i)) matches++;
}
```

3.2 Strongly Connected Components $\mathcal{O}(V + E)$

```
vector<vi> adj; // adjacency matrix
vi index, lowlink; // lowest index reachable
stack<int> tarjanStack;
vector<bool> inStack; // true iff in tarjanStack
int newId;
vector<vi> scc; // Output: collection of vertex sets

void tarjan(int v) {
    index[v] = lowlink[v] = newId++;
    tarjanStack.push(v);
    inStack[v] = true;
    for (int w : adj[v]) {
        if (index[w] == 0) {
            tarjan(w);
            lowlink[v] = min(lowlink[v], lowlink[w]);
        } else if (inStack[w]) {
            lowlink[v] = min(lowlink[v], index[w]);
        }
    }

    if (lowlink[v] == index[v]) {
        scc.push_back(vi());
        int w;
        do {
            w = tarjanStack.top();
            scc.back().push_back(w);
            inStack[w] = false;
            tarjanStack.pop();
        } while (w != v);
    }
}

int findSCC() {
    newId = 1;
    index.clear(); index.resize(n + 1, 0);
    lowlink.clear(); lowlink.resize(n + 1, 0);
    inStack.clear(); inStack.resize(n + 1, false);
    while (!tarjanStack.empty()) tarjanStack.pop();
    scc.clear();

    for (int i = 0; i < n; i++) {
        if (index[i] == 0) tarjan(i);
    }
    return scc.size();
}
```

3.3 Cycle Detection $\mathcal{O}(V + E)$

```
vector<vi> adj; // assumes bidirected graph, adjust accordingly
vector<bool> vis(MAXN, false);
vector<int> par(MAXN, -1);

bool cycle_detection() {
    stack<int> s;
    s.push(0);
```

```
    vis[0] = true;
    while(!s.empty()) {
        int cur = s.top();
        s.pop();
        for(int i : adj[cur]) {
            if(vis[i] && par[cur] != i) return true;
            s.push(i);
            par[i] = cur;
            vis[i] = true;
        }
    }
    return false;
}
```

3.4 Shortest path

3.4.1 BFS $\mathcal{O}(V + E)$

```
int n, dist[MAXN];
vector<int> edges[MAXN]; // (to, cost)

// faster than dijkstra when all edge costs are the same
int bfs(int from, int to) {
fill_n(dist, n, -1);
dist[from] = 0;

queue<int> q;
q.push(from);
while (!q.empty()) {
int cur = q.front();
q.pop();
for (int nxt : edges[cur]) {
if (dist[nxt] >= 0) {
dist[nxt] = dist[cur] + 1;
if (nxt == to) return dist[nxt];
q.push(nxt);
}
}
}
return -1;
}
```

3.4.2 Dijkstra $\mathcal{O}(E + V \log V)$

```
int n; // number of nodes

vector<pii> edges[MAXN]; // (to, cost)
int dist[MAXN];
bool vis[MAXN];

void dijkstra() {
fill_n(vis, n, false);
priority_queue<pii, vector<pii>, greater<pii> > q; // (dist, id)
q.push(pii(0, 0));

while (!q.empty()) {
pii v = q.top();
q.pop();

if (vis[v.second]) continue;
vis[v.second] = true;

for (const pii e : edges[v.second]) {
q.push(pii(v.first + e.second, e.first));
```

```
}
dist[v.second] = v.first;
}
}
```

3.4.3 Floyd-Warshall $\mathcal{O}(V^3)$

```
int n = 100, d[MAXN][MAXN];
for (int i = 0; i < n; i++) fill_n(d[i], n, INF / 3);
// set direct distances from i to j in d[i][j] (and d[j][i])
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
for (int k = 0; k < n; k++)
d[j][k] = min(d[j][k], d[j][i] + d[i][k]);
```

3.4.4 Bellman Ford $\mathcal{O}(VE)$

```
vector< pair<pii,int> > edges; // ((from, to), cost)
vector<int> dist(MAXN);

bool bellman_ford(int source) {
    for (int i = 0; i < MAXN; i++) dist[i] = INF / 3;
    dist[source] = 0;

    bool updated;
    int loops = 0;
    do {
        updated = false;
        for (auto e : edges) {
            int alt = dist[e.first.first] + e.second;
            if (alt < dist[e.first.second]) {
                dist[e.first.second] = alt;
                updated = true;
            }
            // if undirected graph:
            int alt = dist[e.first.second] + e.second;
            if (UNDIRECTED && alt < dist[e.first.first]) {
                dist[e.first.first] = alt;
                updated = true;
            }
        }
    } while(updated && loops < n);
    return loops < n; // loops >= n: negative cycles
}
```

3.5 Max-flow min-cut

3.5.1 Dinic's Algorithm $\mathcal{O}(V^2E)$

```
// http://www.slideshare.net/KuoE0/acmicpc-dinics-algorithm
struct edge {
    int to, rev;
    ll cap, flow;
    edge(int t, int r, ll c) : to(t), rev(r), cap(c), flow(0) {}
};

int s, t, level[MAXN]; // s = source, t = sink
vector<edge> g[MAXN];

bool dinic_bfs() {
    fill_n(level, MAXN, 0);
    level[s] = 1;
```

```

    queue<int> q;
    q.push(s);
    while (!q.empty()) {
        int cur = q.front();
        q.pop();
        for (edge e : g[cur]) {
            if (level[e.to] == 0 && e.flow < e.cap) {
                level[e.to] = level[cur] + 1;
                q.push(e.to);
            }
        }
    }
    return level[t] != 0;
}

ll dinic_dfs(int cur, ll maxf) {
    if (cur == t) return maxf;

    ll f = 0;
    bool isSat = true;
    for (edge &e : g[cur]) {
        if (level[e.to] != level[cur] + 1 || e.flow >= e.cap)
            continue;
        ll df = dinic_dfs(e.to, min(maxf - f, e.cap - e.flow));
        f += df;
        e.flow += df;
        g[e.to][e.rev].flow -= df;
        isSat &= e.flow == e.cap;
        if (maxf == f) break;
    }
    if (isSat) level[cur] = 0;
    return f;
}

ll dinic_maxflow() {
    ll f = 0;
    while (dinic_bfs()) f += dinic_dfs(s, LLINF);
    return f;
}

void add_edge(int fr, int to, ll cap) {
    g[fr].push_back(edge(to, g[to].size(), cap));
    g[to].push_back(edge(fr, g[fr].size() - 1, 0));
}
```

3.6 Min-cost max-flow

```
struct edge {
    // to, rev, flow, capacity, weight
    int t, r;
    ll f, c, w;
    edge(int _t, int _r, ll _c, ll _w) : t(_t), r(_r), f(0), c(_c), w(_w) {}
};

int n, par[MAXN];
vector<edge> adj[MAXN];
ll dist[MAXN];

bool findPath(int s, int t)
{
    fill_n(dist, n, LLINF);
    fill_n(par, n, -1);

    priority_queue< pii, vector<pii>, greater<pii> > q;
    q.push(pii(dist[s] = 0, s));
```

```

    while (!q.empty()) {
        int d = q.top().first, v = q.top().second;
        q.pop();
        if (d > dist[v]) continue;

        for (edge e : adj[v]) {
            if (e.f < e.c && d + e.w < dist[e.t]) {
                q.push(pii(dist[e.t] = d + e.w, e.t));
                par[e.t] = e.r;
            }
        }
    }
    return dist[t] < INF;
}

pair<ll, ll> minCostMaxFlow(int s, int t)
{
    ll cost = 0, flow = 0;
    while (findPath(s, t)) {
        ll f = INF, c = 0;
        int cur = t;
        while (cur != s) {
            const edge &rev = adj[cur][par[cur]], &e = adj[rev.t][rev.r];
            f = min(f, e.c - e.f);
            cur = rev.t;
        }
        cur = t;
        while (cur != s) {
            edge &rev = adj[cur][par[cur]], &e = adj[rev.t][rev.r];
            c += e.w;
            e.f += f;
            rev.f -= f;
            cur = rev.t;
        }
        cost += f * c;
        flow += f;
    }
    return pair<ll, ll>(cost, flow);
}

inline void addEdge(int from, int to, ll cap, ll weight)
{
    adj[from].push_back(edge(to, adj[to].size(), cap, weight));
    adj[to].push_back(edge(from, adj[from].size() - 1, 0, -weight));
}
```

3.7 Minimal Spanning Tree

3.7.1 Prim $\mathcal{O}((E+V)\log V)$

```
// minimum spanning forest actually...
vector<pii> edges[MAXN]; // or set
int dist[MAXN];
bool done[MAXN];

ll prim(int n) {
    fill_n(dist, n, INF);
    fill_n(done, n, false);
    ll ret = 0, trees = 0;
    set<pii> q; // (to MST, vertex)
    for (int i = 0; i < n; i++) {
        if (done[i]) continue;
        trees++;
        q.insert(pii(dist[i] = 0, i));
        while (!q.empty()) {
```

```
            ret += q.begin()->first;
            int cur = q.begin()->second;
            q.erase(q.begin());
            done[cur] = true;
            for (pii pr : edges[cur]) {
                if (!done[pr.first] && pr.second < dist[pr.first]) {
                    q.erase(pii(dist[pr.first], pr.first));
                    dist[pr.first] = pr.second;
                    q.insert(pii(dist[pr.first], pr.first));
                }
            }
        }
    }
    // if (trees > 1) return -1; // forest
    return ret;
}
```

3.7.2 Kruskal $\mathcal{O}(E \log V)$

```
struct edge {
    int x, y, s;
    void read() { cin >> x >> y >> s; }
};

edge edges[MAXM];

int kruskal(int n, int m) {
    uf_init(n);
    sort(edges, edges + m, [] (const edge &a, const edge &b)
        -> bool { return a.s > b.s; });
    ll ret = 0;
    while (m--) {
        if (uf_find(edges[m].x) != uf_find(edges[m].y)) {
            ret += edges[m].s;
            uf_union(edges[m].x, edges[m].y);
        }
    }
    return ret;
}
```

4 String algorithms

4.1 Z-algorithm $\mathcal{O}(n)$

```
// z[i] = length of longest substring starting from s[i],
// which is also a prefix of s.
vector<int> z_function(const string &s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```

4.2 Longest Common Subsequence $\mathcal{O}(n^2)$

SUBSTRING: *consecutive characters* !!!

```
int table[STR_SIZE][STR_SIZE]; // DP problem

int lcs(const string &w1, const string &w2) {
    int n1 = w1.size(), n2 = w2.size();
    for (int i = 0; i <= n1; i++) table[i][0] = 0;
    for (int j = 0; j <= n2; j++) table[0][j] = 0;

    for (int i = 1; i < n1; i++) {
        for (int j = 1; j < n2; j++) {
            table[i][j] = w1[i - 1] == w2[j - 1] ?
                (table[i - 1][j - 1] + 1) :
                max(table[i - 1][j], table[i][j - 1]);
        }
    }
    return table[n1][n2];
}

// backtrace
string getLCS(const string &w1, const string &w2) {
    int i = w1.size(), j = w2.size();
    string ret = "";
    while (i > 0 && j > 0) {
        if (w1[i - 1] == w2[j - 1]) ret += w1[--i], j--;
        else if (table[i][j - 1] > table[i - 1][j]) j--;
        else i--;
    }
    reverse(ret.begin(),ret.end());
    return ret;
}
```

4.3 Levenshtein Distance $\mathcal{O}(n^2)$

```
int costs[MAX_SIZE][MAX_SIZE]; // DP problem

int levDist(const string &w1, const string &w2) {
    int n1 = w1.size(), n2 = w2.size();
    for (int i = 0; i <= n1; i++) costs[i][0] = i; // removal
    for (int j = 0; j <= n2; j++) costs[0][j] = j; // insertion
    for (int i = 1;i <= n1; i++) {
        for (int j = 1; j <= n2; j++) {
            costs[i][j] = min(
                min(costs[i - 1][j] + 1, costs[i][j - 1] + 1),
                costs[i - 1][j - 1] + (w1[i - 1] != w2[j - 1])
            );
        }
    }
    return costs[n1][n2];
}
```

4.4 Knuth-Morris-Pratt algorithm $\mathcal{O}(N + M)$

```
int kmp_search(const string &word, const string &text) {
    int n = word.size();
    vector<int> table(n + 1, 0);
    for (int i = 1, j = 0; i < n; ) {
        if (word[i] == word[j]) {
            table[++i] = ++j;
        } else if (j > 0) {
            j = table[j];
        } else i++;
```

```
    }
    int matches = 0;
    for (int i = 0, j = 0; i < text.size(); ) {
        if (text[i] == word[j]) {
            i++;
            if (++j == n) {
                matches++;
                // match at interval [i - j, i)
                j = table[j];
            }
        } else if (j > 0) j = table[j];
        else i++;
    }
    return matches;
}
```

4.5 Aho-Corasick Algorithm $\mathcal{O}(N + \sum_{i=1}^{m} |S_i|)$

```

const int MAXP = 100, MAXLEN = 200, SIGMA = 26, MAXTRIE = MAXP * MAXLEN;

int npatterns;
string patterns[MAXP], S;

int wordIdx[MAXTRIE], to[MAXTRIE][SIGMA], sLink[MAXTRIE], dLink[MAXTRIE], nnodes;

void ahoCorasick()
{
    // 1. Make a tree, 2. create sLinks and dLinks, 3. Walk through S

    fill_n(wordIdx, MAXTRIE, -1);
    for (int i = 0; i < MAXTRIE; i++) fill_n(to[i], SIGMA, 0);
    fill_n(sLink, MAXTRIE, 0);
    fill_n(dLink, MAXTRIE, 0);
    nnodes = 1;

    for(int i = 0; i < npatterns; i++) {
        int cur = 0;
        for (char c : patterns[i]) {
            int idx = c - 'a';
            if(to[cur][idx] == 0) to[cur][idx] = nnodes++;
            cur = to[cur][idx];
        }
        wordIdx[cur] = i;
    }

    queue<int> q;
    q.push(0);
    while(!q.empty()) {
        int cur = q.front(); q.pop();
        for(int c = 0; c < SIGMA; c++) {
            if(to[cur][c]) {
                int sl = sLink[to[cur][c]] = cur == 0 ? 0 : to[sLink[cur]][c];
                // if all strings have equal length, remove this:
                dLink[to[cur][c]] = wordIdx[sl] >= 0 ? sl : dLink[sl];
                q.push(to[cur][c]);
            } else to[cur][c] = to[sLink[cur]][c];
        }
    }

    for (int cur = 0, i = 0, n = S.size(); i < n; i++) {
        int idx = S[i] - 'a';
        cur = to[cur][idx];
        // we have a match! (if g[i][j] >= 0)
        for (int hit = wordIdx[cur] >= 0 ? cur : dLink[cur]; hit; hit = dLink[hit]) {
```

```
            cerr << "Match for " << patterns[wordIdx[hit]] << " at " << (i + 1 - patterns[
                wordIdx[hit]].size()) << endl;
        }
    }
}
```

5 Geometry

```
typedef double NUM; // either double or long long

struct pt {
    NUM x, y;

    pt() : x(0), y(0) {}
    pt(NUM _x, NUM _y) : x(_x), y(_y) {}
    pt(const pt &p) : x(p.x), y(p.y) {}

    pt operator*(NUM scalar) const {
        return pt(scalar * x, scalar * y); // scalar
    }
    NUM operator*(const pt &rhs) const {
        return x * rhs.x + y * rhs.y; // dot product
    }
    NUM operator^(const pt &rhs) const {
        return x * rhs.y - y * rhs.x; // cross product
    }
    pt operator+(const pt &rhs) const {
        return pt(x + rhs.x, y + rhs.y); // addition
    }
    pt operator-(const pt &rhs) const {
        return pt(x - rhs.x, y - rhs.y); // subtraction
    }
    bool operator==(const pt &rhs) const {
        return x == rhs.x && y == rhs.y;
    }
    bool operator!=(const pt &rhs) const {
        return x != rhs.x || y != rhs.y;
    }
};

// distance SQUARED from pt a to pt b
NUM sqDist(const pt &a, const pt &b) {
    return (a - b) * (a - b);
}

// distance SQUARED from pt a to line bc
double sqDistPointLine(pt a, pt b, pt c) {
    a = a - b;
    c = c - b;
    return (a ^ c) * (a ^ c) / (double)(c * c);
}

// distance SQUARED from pt a to line segment c
double sqDistPointSegment(pt a, pt b, pt c) {
    a = a - b;
    c = c - b;
    NUM dot = a * c, len = c * c;
    if (dot <= 0) return a * a;
    if (dot >= len) return (a - c) * (a - c);
    return a * a - dot * dot / ((double) len);
    // pt proj = c * dot / ((double) len);
}

bool between(NUM a, NUM b, NUM n) {
    return min(a, b) <= n && n <= max(a, b);
```

```
}
bool collinear(pt a, pt b, pt c) {
    return (a - b) ^ (a - c) == 0;
}

// point a on segment bc
bool pointOnSegment(pt a, pt b, pt c)
{
    return collinear(a, b, c) &&
        between(b.x, c.x, a.x) && between(b.y, c.y, a.y);
}

pt lineLineIntersection(pt a, pt b, pt c, pt d, bool &cross)
{
    pt res = (c - d) * (a ^ b) - (a - b) * (c ^ d);
    NUM det = (a.x - b.x) * (c.y - d.y) - (a.y - b.y) * (c.x - d.x);
    cross = det != 0;
    if (cross) res = res / det;
    return res;
}

// Line segment a1 -- a2 intersects with b1 -- b2?
// returns 0: no, 1: yes at i1, 2: yes at i1 -- i2
int segmentsIntersect(pt a1, pt a2, pt b1, pt b2, pt &i1, pt &i2) {
    if (((a2 - a1) ^ (b2 - b1)) < 0) swap(a1, a2);
    // assert(a1 != a2 && b1 != b2);
    pt q = a2 - a1, r = b2 - b1, s = b1 - a1;
    NUM cross = q ^ r, c1 = s ^ r, c2 = s ^ q;
    if (cross == 0) {
        // line segments are parallel
        if ((q ^ s) != 0) return 0; // no intersection
        NUM v1 = s * q, v2 = (b2 - a1) * q, v3 = q * q;
        if (v2 < v1) swap(v1, v2), swap(b1, b2);

        if (v1 > v3 || v2 < 0) return 0; // intersection empty
        i1 = v2 > v3 ? a2 : b2;
        i2 = v1 < 0 ? a1 : b1;
        return i1 == i2 ? 1 : 2; // one point or overlapping
    } else { // cross > 0
        i1 = pt(a1) + pt(q) * (1.0 * c1 / cross); // needs double
        return 0 <= c1 && c1 <= cross && 0 <= c2 && c2 <= cross;
        // intersection inside segments
    }
}

// complete intersection check
int segmentsIntersect2(pt a1, pt a2, pt b1, pt b2, pt &i1, pt &i2) {
    if (a1 == a2 && b1 == b2) {
        i1 = a1;
        return a1 == b1;
    } else if (a1 == a2) {
        i1 = a1;
        return pointOnSegment(a1, b1, b2);
    } else if (b1 == b2) {
        i1 = b1;
        return pointOnSegment(b1, a1, a2);
    } else return segmentsIntersect(a1, a2, b1, b2, i1, i2);
}

// Returns TWICE the area of a polygon to keep it an integer
NUM polygonTwiceArea(const vector<pt> &polygon) {
    NUM area = 0;
    for (int i = 0, N = polygon.size(), j = N - 1; i < N; j = i++)
        area += polygon[i] ^ polygon[j];
    return abs(area);
}

// returns 0 outside, 1 inside, 2 on boundary
```

```cpp
int pointInPolygon(pt p, const vector<pt> &polygon) {
    // Check corssings with horizontal semi-line through p to +x
    int crosscount = 0, N = polygon.size();
    for (int i = 0, j = N - 1; i < N; j = i++) {
        if (pointOnSegment(p, polygon[i], polygon[j])) return 2;

        // check if it crosses the vertical y = p.y line
        NUM l = (p.x - polygon[i].x)*(polygon[j].y - polygon[i].y);
        NUM r = (p.y - polygon[i].y)*(polygon[j].x - polygon[i].x);
        if (polygon[j].y > p.y) {
            if (polygon[i].y <= p.y && l < r) crosscount++;
        } else {
            if (polygon[i].y <= p.y && l > r) crosscount++;
        }
    }
    return crosscount & 1;
}

// Assumption: polygon has unique points
int pointInConvex(pt p, const vector<pt> &polygon) {
    // the cross product should always have the same sign,
    // when the point is inside the convex
    int N = polygon.size(), sgn = 0;
    bool onBoundary = false;
    for (int i = 0, j = N - 1; i < N; j = i++) {
        NUM cross = (polygon[j] - p) ^ (polygon[i] - p);
        if (cross == 0) onBoundary = true;
        else if (sgn == 0) sgn = sign(cross);
        else if (sgn != sign(cross)) return 0;
    }
    return onBoundary ? 2 : 1;
}
```

5.1 Convex Hull $\mathcal{O}(n \log n)$

```cpp
// output contains indices of the points on the hull
void convex_hull(const vector<pt> &pts, vector<int> &output) {
    output.clear();
    if (pts.size() < 3) {
        if (pts.size() >= 1) output.push_back(0);
        if (pts.size() >= 2) output.push_back(1);
        return;
    }

    unsigned int bestIndex = 0;
    NUM minX = pts[0].x, minY = pts[0].y;
    for(unsigned int i = 1; i < pts.size(); ++i) {
        if (pts[i].x < minX || (pts[i].x == minX && pts[i].y < minY)) {
            bestIndex = i;
            minX = pts[i].x;
            minY = pts[i].y;
        }
    }
    vector<int> ordered; //index into pts
    for(unsigned int i = 0; i < pts.size(); ++i) {
        if (i != bestIndex) ordered.push_back(i);
    }

    pt refr = pts[bestIndex];
    sort(ordered.begin(), ordered.end(), [&pts,&refr] (int a, int b) -> bool {
        NUM cross = (pts[a] - refr) ^ (pts[b] - refr);
        return cross != 0 ? cross > 0 : sqDist(refr, pts[a]) < sqDist(refr, pts[b]);
    });

    output.push_back(bestIndex);
    output.push_back(ordered[0]);
```

```
    output.push_back(ordered[1]);
    for(unsigned int i = 2; i < ordered.size(); ++i) {
        //NOTE: > INCLUDES and >= EXCLUDES points on the hull-line
        while (output.size() > 1 && ((pts[output[output.size() - 2]] - pts[output.back()]) ^ (
            pts[ordered[i]] - pts[output.back()])) > 0) {
            output.pop_back();
        }
        output.push_back(ordered[i]);
    }
    return;
}
```

6 Miscellaneous

6.1 Binary search $\mathcal{O}(\log n)$

Inclusive, Exclusive

```
bool test(int n);

int min = 0, max = n;
// assert(test(min) && !test(max));
while (max - min > 1) {
    int c = (min + max) / 2;
    if (test(c)) min = c;
    else         max = c;
}
return min;
```

Inclusive, Inclusive

```
bool test(int n);

int lo = 0, hi = n - 1;
// assert(test(lo) && !test(hi + 1));
while (lo < hi) {
    int mid = (lo + hi + 1) / 2;
    if (test(mid)) lo = mid;
    else hi = mid - 1;
}
return lo;
```

6.2 Fast Fourier Transform $\mathcal{O}(n \log n)$

Given two polynomials $A(x) = a_0 + a_1 x + \ldots + a_{n/2} x^{n/2}$ and $B(x) = b_0 + b_1 x + \ldots + b_{n/2} x^{n/2}$, FFT calculates all coefficients of $C(x) = A(x) \cdot B(x) = c_0 + c_1 x + \ldots c_n x^n$.

```

typedef complex<double> cpx;
const int logmaxn = 20, maxn = 1 << logmaxn;

cpx a[maxn] = {}, b[maxn] = {}, c[maxn];

void fft(cpx *src, cpx *dest)
{
    for (int i = 0, rep = 0; i < maxn; i++, rep = 0) {
        for (int j = i, k = logmaxn; k--; j >>= 1) rep = (rep << 1) | (j & 1);
        dest[rep] = src[i];
    }
    for (int s = 1, m = 1; m <= maxn; s++, m *= 2) {
        cpx r = exp(cpx(0, 2.0 * pi / m));
        for (int k = 0; k < maxn; k += m) {
            cpx cr(1.0, 0.0);
            for (int j = 0; j < m / 2; j++) {
                NUM t = cr * dest[k + j + m / 2];
                dest[k + j + m / 2] = dest[k + j] - t;
                dest[k + j] += t;
                cr *= r;
            }
        }
    }
}

void multiply()
{
    fft(a, c);
```

```
    fft(b, a);
    for (int i = 0; i < maxn; i++) b[i] = conj(a[i] * c[i]);
    fft(b, c);
    for (int i = 0; i < maxn; i++) c[i] = conj(c[i]) / (1.0 * maxn);
}
```

6.3 Minimum Assignment (Hungarian Algorithm) $\mathcal{O}(n^3)$

```
int n, m; // n rows, m columns
int a[MAXN + 1][MAXM + 1]; // matrix, 1-based
int minimum_assignment() {
    vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);

    for (int i = 1; i <= n; i++) {
        p[0] = i;
        int j0 = 0;
        vector<int> minv(m + 1, INF);
        vector<char> used(m + 1, false);
        do {
            used[j0] = true;
            int i0 = p[j0], delta = INF, j1;
            for (int j = 1; j <= m; j++)
                if (!used[j]) {
                    int cur = a[i0][j] - u[i0] - v[j];
                    if (cur < minv[j]) minv[j] = cur, way[j] = j0;
                    if (minv[j] < delta) delta = minv[j], j1 = j;
                }
            for (int j = 0; j <= m; ++ j) {
                if(used[j]) u[p[j]] += delta, v[j] -= delta;
                else minv[j] -= delta;
            }
            j0 = j1;
        } while (p[j0] != 0);
        do {
            int j1 = way[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }

//  for (int j = 1; j <= m; ++ j) ans[p[j]] = j;
    return -v[0];
}
```