

IME ACM-ICPC Team Notebook

Contents

1 Template + vimrc

1.1	Template	1
1.2	vimrc	2

2 Graphs

2.1	Toposort	2
2.2	Articulation points and bridges	2
2.3	Strongly Connected Components	2
2.4	Tarjan	2
2.5	MST (Kruskal)	3
2.6	MST (Prim)	3
2.7	Shortest Path (SPFA)	3
2.8	Shortest Path (Floyd-Warshall)	3
2.9	Max Flow	3
2.10	Min Cost Max Flow	3
2.11	Max Bipartite Cardinality Matching (Kuhn)	4
2.12	Lowest Common Ancestor	4
2.13	2-SAT	4
2.14	Erdos-Gallai	5
2.15	Block Cut	5
2.16	Stoer Wagner (Stanford)	5

3 Mathematics

3.1	Basics	5
3.2	Sieve of Eratosthenes	6
3.3	Euler Phi	6
3.4	Extended Euclidean	6
3.5	Miller-Rabin	6
3.6	Prime Factors	6
3.7	Pollard-Rho	6
3.8	Fast Fourier Transform	7
3.9	Fast Fourier Transform(Tourist)	7
3.10	Number Theoretic Transform	8
3.11	Fast Walsh-Hadamard Transform	9
3.12	Chinese Remainder	9
3.13	Primitive Root	9
3.14	Gaussian Elimination (double)	9
3.15	Gaussian Elimination (modulo prime)	10
3.16	Gaussian Elimination (extended inverse)	10
3.17	Golden Section Search (Ternary Search)	10
3.18	Josephus	10
3.19	Simpson Rule	10
3.20	Discrete Log (Baby-step Giant-step)	10
3.21	Mobius Function	11
3.22	Simplex (Stanford)	11

4 Strings

4.1	Rabin-Karp	11
4.2	Knuth-Morris-Pratt	12
4.3	Z Function	12
4.4	Prefix Function	12
4.5	Recursive-String Matching	12
4.6	Aho-Corasick	12
4.7	Manacher	13
4.8	Suffix Array	13
4.9	Suffix Automaton	14
4.10	Z Function (Stanford)	14

5 Data Structures

5.1	Disjoint Set Union	15
5.2	Sparse Table	15

5.3	Sparse Table 2D	15
5.4	Fenwick Tree	15
5.5	Fenwick Tree 2D	15
5.6	Range Update Point Query Fenwick Tree	15
5.7	Segment Tree	16
5.8	Segment Tree 2D	16
5.9	Persistent Segment Tree	16
5.10	Persistent Segment Tree (Naum)	17
5.11	Heavy-Light Decomposition	17
5.12	Heavy-Light Decomposition (new)	17
5.13	Centroid Decomposition	17
5.14	Trie	18
5.15	Mergesort Tree	18
5.16	Treap	18
5.17	KD Tree (Stanford)	19
5.18	Maximum Queue	20
5.19	Ordered Set	20
5.20	Lichao Tree (ITA)	20

6 Dynamic Programming

6.1	Longest Increasing Subsequence	20
6.2	Convex Hull Trick	21
6.3	Convex Hull Trick (emaxx)	21
6.4	Divide and Conquer Optimization	21
6.5	Knuth Optimization	21

7 Geometry

7.1	Basic	22
7.2	Basic (new)	23
7.3	Closest Pair of Points	24
7.4	Nearest Neighbours	25

8 Geometry (Stanford)

8.1	Basic	25
8.2	Convex Hull	27
8.3	Delaunay Triangulation	27

9 Geometry (ITA)

9.1	Circulo 2d	28
9.2	Minkowski	29
9.3	Great Circle	30

10 Miscellaneous

10.1	builtin	30
10.2	prime numbers	30
10.3	Week day	30
10.4	Date	30
10.5	Latitude Longitude (Stanford)	31
10.6	Python	31
10.7	Sqrt Decomposition	31
10.8	Bitset	31
10.9	Parenthesis to Polish (ITA)	31

11 Math Extra

11.1	Combinatorial formulas	32
11.2	Number theory identities	32
11.3	Stirling Numbers of the second kind	32
11.4	Burnside's Lemma	32
11.5	Numerical integration	32

1 Template + vimrc

1.1 Template

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

#define st first
#define nd second
#define mp make_pair
#define pb push_back
#define cl(x, v) memset((x), (v), sizeof(x))
#define gcd(x, y) __gcd((x), (y))

#ifdef ONLINE_JUDGE
#define db(x) cerr << #x << " == " << x << endl
#define dbs(x) cerr << x << endl
#define _ << " ", " <<
#else
#define db(x) ((void)0)
#define dbs(x) ((void)0)
#endif

typedef long long ll;
typedef long double ld;

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

const ld EPS = 1e-9, PI = acos(-1.);
const int INF = 0x3f3f3f3f, MOD = 1e9+7;
const int N = 1e5+5;

int main() {
    //freopen("in", "r", stdin);
    //freopen("out", "w", stdout);
    return 0;
}
```

1.2 vimrc

```
syntax on
set et ts=2 sw=0 sts=-1 ai nu hls cindent
nnoremap ; :
vnoremap ; :
noremap <c-j> 15gj
noremap <c-k> 15gk
nnoremap <s-k> i<CR><ESC>
inoremap ,. <esc>
vnoremap ,. <esc>
nnoremap ,. <esc>
```

2 Graphs

2.1 Toposort

```
// Kahn - Topological Sort O(V+E)

vi adj[N];
int inc[N]; // number of incoming arcs/edges

queue<int> q;
for (int i = 1; i <= n; ++i) if (inc[i]<=1) q.push(i);

while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int v : adj[u])
        // undirected graph: inc[v] <= 1
        // directed graph: inc[v] == 0
        if (inc[v] > 1 and --inc[v] <= 1)
            q.push(v);
}
```

2.2 Articulation points and bridges

```
// Articulation points and Bridges O(V+E)
int par[N], art[N], low[N], num[N], ch[N], cnt;

void articulation(int u) {
    low[u] = num[u] = ++cnt;
    for (int v : adj[u]) {
        if (!num[v]) {
            par[v] = u; ch[u]++;
            articulation(v);
            if (low[v] >= num[u]) art[u] = 1;
            if (low[v] > num[u]) // u-v bridge
                low[u] = min(low[u], low[v]);
        }
        else if (v != par[u]) low[u] = min(low[u], num[v]);
    }
}

for (int i = 0; i < n; ++i) if (!num[i])
    articulation(i), art[i] = ch[i]>1;
```

2.3 Strongly Connected Components

```
// Kosaraju - SCC O(V+E)
vi adj[N], adjt[N];
int ord[N], ordn, scc[N], sccn, vis[N];

void dfs(int u) {
    vis[u] = 1;
    for (int v : adj[u]) if (!vis[v]) dfs(v);
    ord[ordn++] = u;
}

void dfst(int u) {
    vis[u] = 0;
    for (int v : adjt[u]) if (vis[v]) dfst(v);
    scc[u] = sccn;
}

// in main
sccn = ordn = 1;
for (int i = 1; i <= n; ++i) if (!vis[i]) dfs(i);
for (int i = n; i > 0; --i) if (vis[ord[i]]) dfst(ord[i]), sccn++;
```

2.4 Tarjan

```
// Tarjan for SCC and Edge Biconnected Componentes - O(n + m)
vector<int> adj[N];
stack<int> st;
bool inSt[N];

int low[N], id[N], cmp[N];
int cnt, cmpCnt;

void tarjan(int n){
    id[n] = low[n] = ++cnt;
    st.push(n), inSt[n] = true;

    for(auto x : adj[n]){
        if(id[x] and inSt[x]) low[n] = min(low[n], id[x]);
        else if(!id[x]) {
            tarjan(x);
            if(inSt[x])
                low[n] = min(low[n], low[x]);
        }
    }

    if(low[n] == id[n]){
        while(st.size()){
            int x = st.top();
            inSt[x] = false;
            cmp[x] = cmpCnt;

            st.pop();
            if(x == n) break;
        }
        cmpCnt++;
    }
}
```

2.5 MST (Kruskal)

```
// Kruskal - MST  $O(E \log E)$ 
vector<pii> edges;
// + Union-find

sort(edges.begin(), edges.end());
int cost = 0;
for (auto e : edges)
    if (find(e.nd.st) != find(e.nd.nd))
        unite(e.nd.st, e.nd.nd), cost += e.st;
```

2.6 MST (Prim)

```
// Prim - MST  $O(E \log E)$ 
vi adj[N], adjw[N];
int vis[N];

priority_queue<pii> pq;
pq.push(mp(0, 0));

while (!pq.empty()) {
    int u = pq.top().nd;
    pq.pop();
    if (vis[u]) continue;
    vis[u]=1;
    for (int i = 0; i < adj[u].size(); ++i) {
        int v = adj[u][i];
        int w = adjw[u][i];
        if (!vis[v]) pq.push(mp(-w, v));
    }
}
```

2.7 Shortest Path (SPFA)

```
// Shortest Path Faster Algorithm  $O(VE)$ 
int dist[N], inq[N];

cl(dist, 63);
queue<int> q;
q.push(0); dist[0] = 0; inq[0] = 1;

while (!q.empty()) {
    int u = q.front(); q.pop(); inq[u]=0;
    for (int i = 0; i < adj[u].size(); ++i) {
        int v = adj[u][i], w = adjw[u][i];
        if (dist[v] > dist[u] + w) {
            dist[v] = dist[u] + w;
            if (!inq[v]) q.push(v), inq[v] = 1;
        }
    }
}
```

2.8 Shortest Path (Floyd-Warshall)

```
// Floyd-Warshall (APSP)  $O(V^3)$ 
int adj[N][N]; // no-edge = INF

for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
```

2.9 Max Flow

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

const int N = 1e5+1, INF = 1e9;
struct edge {int v, c, f;};

int n, src, snk, h[N], ptr[N];
vector<edge> eds;
vector<int> g[N];

void add_edge (int u, int v, int c) {
    int k = eds.size();
    eds.push_back({v, c, 0});
    eds.push_back({u, 0, 0});
    g[u].push_back(k);
    g[v].push_back(k+1);
}

bool bfs() {
    memset(h, 0, sizeof h);
    queue<int> q;
    h[src] = 1;
    q.push(src);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int i : g[u]) {
            int v = eds[i].v;
            if (!h[v] and eds[i].f < eds[i].c)
                q.push(v), h[v] = h[u] + 1;
        }
    }
    return h[snk];
}

int dfs (int u, int flow) {
    if (!flow or u == snk) return flow;
    for (int &i = ptr[u]; i < g[u].size(); ++i) {
        edge &dir = eds[g[u][i]], &rev = eds[g[u][i]^1];
        int v = dir.v;
        if (h[v] != h[u] + 1) continue;
        int inc = min(flow, dir.c - dir.f);
        inc = dfs(v, inc);
        if (inc) {
            dir.f += inc, rev.f -= inc;
            return inc;
        }
    }
    return 0;
}

int dinic() {
    int flow = 0;
    while (bfs()) {
        memset(ptr, 0, sizeof ptr);
        while (int inc = dfs(src, INF)) flow += inc;
    }
    return flow;
}
```

2.10 Min Cost Max Flow

```
// w: weight or cost, c : capacity
struct edge {int v, f, w, c; };

int node_count, flw_lmt=INF, src, snk, flw, cst, p[N], d[N], et[N];
vector<edge> e;
vector<int> g[N];

void add_edge(int u, int v, int w, int c) {
    int k = e.size();
    node_count = max(node_count, u+1);
    node_count = max(node_count, v+1);
    g[u].push_back(k);
    g[v].push_back(k+1);
    e.push_back({ v, 0, w, c });
    e.push_back({ u, 0, -w, 0 });
}

void clear() {
    flw_lmt = INF;
    for(int i=0; i<node_count; ++i) g[i].clear();
    e.clear();
    node_count = 0;
}
```

```

void min_cost_max_flow() {
    flw = 0, cst = 0;
    while (flw < flw_lmt) {
        memset(et, 0, node_count * sizeof et[0]);
        memset(d, 63, node_count * sizeof d[0]);
        deque<int> q;
        q.push_back(src), d[src] = 0;

        while (!q.empty()) {
            int u = q.front(); q.pop_front();
            et[u] = 2;

            for(int i : g[u]) {
                edge &dir = e[i];
                int v = dir.v;
                if (dir.f < dir.c and d[u] + dir.w < d[v]) {
                    d[v] = d[u] + dir.w;
                    if (et[v] == 0) q.push_back(v);
                    else if (et[v] == 2) q.push_front(v);
                    et[v] = 1;
                    p[v] = i;
                }
            }
        }

        if (d[snk] > INF) break;

        int inc = flw_lmt - flw;
        for (int u=snk; u != src; u = e[p[u]^1].v) {
            edge &dir = e[p[u]];
            inc = min(inc, dir.c - dir.f);
        }

        for (int u=snk; u != src; u = e[p[u]^1].v) {
            edge &dir = e[p[u]], &rev = e[p[u]^1];
            dir.f += inc;
            rev.f -= inc;
            cst += inc * dir.w;
        }

        if (!inc) break;
        flw += inc;
    }
}

```

2.11 Max Bipartite Cardinality Matching (Kuhn)

```

// Kuhn - Maximum Cardinality Bipartite Matching (MCBM) O(VE)
// TIP: If too slow, shuffle nodes and try again.
int x, vis[N], b[N], ans;

bool match(int u) {
    if (vis[u] == x) return 0;
    vis[u] = x;
    for (int v : adj[u])
        if (!b[v] or match(b[v])) return b[v]=u;
    return 0;
}

for (int i = 1; i <= n; ++i) ++x, ans += match(i);

// Maximum Independent Set on bipartite graph
MIS + MCBM = V

// Minimum Vertex Cover on bipartite graph
MVC = MCBM

```

2.12 Lowest Common Ancestor

```

// Lowest Common Ancestor <O(nlogn), O(logn)>
const int N = 1e6, M = 25;
int anc[M][N], h[N], rt;

// TODO: Calculate h[u] and set anc[0][u] = parent of node u for each u

// build (sparse table)
anc[0][rt] = rt; // set parent of the root to itself
for (int i = 1; i < M; ++i)
    for (int j = 1; j <= n; ++j)
        anc[i][j] = anc[i-1][anc[i-1][j]];

```

```

// query
int lca(int u, int v) {
    if (h[u] < h[v]) swap(u, v);
    for (int i = M-1; i >= 0; --i) if (h[u] - (1<<i) >= h[v])
        u = anc[i][u];

    if (u == v) return u;

    for (int i = M-1; i >= 0; --i) if (anc[i][u] != anc[i][v])
        u = anc[i][u], v = anc[i][v];
    return anc[0][u];
}

```

2.13 2-SAT

```

// 2-SAT - O(V+E)
// For each variable x, we create two nodes in the graph: u and !u
// If the variable has index i, the index of u and !u are: 2*i and 2*i+1
int ans[N], cn, vis[N], sccn, scc[N], scch[N];
vector<int> stk, adj[N], radj[N];

void koj(int u) {
    vis[u] = 1;
    for (int v : adj[u]) if (!vis[v]) koj(v);
    stk.pb(u);
}

void rkoj(int u) {
    vis[u] = 0;
    for (int v : radj[u]) if (vis[v]) rkoj(v);
    scc[u] = sccn;
}

void rsat(int);

void sat(int u) {
    if (ans[u] != -1) return;
    ans[u] = 1, rsat(u^1);
    for (int v : adj[u]) sat(v);
}

void rsat(int u) {
    if (ans[u] != -1) return;
    ans[u] = 0, sat(u^1);
    for (int v : radj[u]) rsat(v);
}

bool sat() {
    sccn = 1; // Kosaraju's
    for (int i = 2; i <= cn; i++) if (!vis[i]) koj(i);
    while (stk.size()) {
        int u = stk.back(); stk.pop_back();
        if (vis[u]) rkoj(u), scch[sccn++] = u;
    }

    // Checks if exists a valid assingment.
    for (int i = 2; i <= cn; i+=2) if (scc[i] == scc[i^1]) return 0;

    memset(ans, -1, sizeof ans);
    // To set value of x to true (or false) call sat(x) (or sat(x^1)) below.

    // Finds a valid assignment. Erase if not needed.
    for(int i = sccn - 1; i > 0; --i) sat(scch[i]);

    // Check for inconsistencies in implication graph.
    for (int u = 2; u <= cn; u++) if (ans[u] == 1)
        for (int v : adj[u]) if (!ans[v]) return 0;

    return 1;
}

void add_clause(int x, int y) {
    cn = max(cn, max(x^1, y^1));
    adj[x].pb(y), adj[y^1].pb(x^1);
    radj[y].pb(x), radj[x^1].pb(y^1);
}

void reset() {
    memset(ans, -1, sizeof ans);
    for(int i=0; i<=cn; ++i) adj[i].clear(), radj[i].clear();
    cn = 0;
}

```

2.14 Erdos-Gallai

```
// Erdos-Gallai - O(nlogn)
// check if it's possible to create a simple graph (undirected edges) from
// a sequence of vertex's degrees
bool gallai(vector<int> v) {
    vector<ll> sum;
    sum.resize(v.size());

    sort(v.begin(), v.end(), greater<int>());
    sum[0] = v[0];
    for (int i = 1; i < v.size(); i++) sum[i] = sum[i-1] + v[i];
    if (sum.back() % 2) return 0;

    for (int k = 1; k < v.size(); k++) {
        int p = lower_bound(v.begin(), v.end(), k, greater<int>()) - v.begin();
        if (p < k) p = k;
        if (sum[k-1] > 1ll*k*(p-1) + sum.back() - sum[p-1]) return 0;
    }
    return 1;
}
```

2.15 Block Cut

```
// Tarjan for Block Cut Tree (Node Biconnected Componentes) - O(n + m)
#define pb push_back
#include <bits/stdc++.h>
using namespace std;

const int N = 1e5+5;

// Regular Tarjan stuff
int n, num[N], low[N], cnt, ch[N], art[N];
vector<int> adj[N], st;

int lb[N]; // Last block that node is contained
int bn; // Number of blocks
vector<int> blc[N]; // List of nodes from block

void dfs(int u, int p) {
    num[u] = low[u] = ++cnt;
    ch[u] = adj[u].size();
    st.pb(u);

    if (adj[u].size() == 1) blc[++bn].pb(u);

    for(int v : adj[u]) {
        if (!num[v]) {
            dfs(v, u, low[u] = min(low[u], low[v]));
            if (low[v] == num[u]) {
                if (p != -1 or ch[u] > 1) art[u] = 1;
                blc[++bn].pb(u);
                while(blc[bn].back() != v)
                    blc[bn].pb(st.back()), st.pop_back();
            }
        }
        else if (v != p) low[u] = min(low[u], num[v]), ch[v]--;
    }

    if (low[u] == num[u]) st.pop_back();
}

// Nodes from 1 .. n are blocks
// Nodes from n+1 .. 2*n are articulations
vector<int> bct[2*N]; // Adj list for Block Cut Tree

void build_tree() {
    for(int u=1; u<=n; ++u) for(int v : adj[u]) if (num[u] > num[v]) {
        if (lb[u] == lb[v] or blc[lb[u]][0] == v) /* edge u-v belongs to block lb[u] */;
        else /* edge u-v belongs to block cut tree */;
        int x = (art[u] ? u + n : lb[u]), y = (art[v] ? v + n : lb[v]);
        bct[x].pb(y), bct[y].pb(x);
    }
}

void tarjan() {
    for(int u=1; u<=n; ++u) if (!num[u]) dfs(u, -1);
    for(int b=1; b<=bn; ++b) for(int u : blc[b]) lb[u] = b;
    build_tree();
}
```

2.16 Stoer Wagner (Stanford)

```
// a is a N*N matrix storing the graph we use; a[i][j]=a[j][i]
memset(use,0,sizeof(use));
ans=MAXLONGINT;
for (int i=1;i<N;i++)
{
    memcpy(visit,use,505*sizeof(int));
    memset(reach,0,sizeof(reach));
    memset(last,0,sizeof(last));
    t=0;
    for (int j=1;j<=N;j++)
        if (use[j]==0) {t=j;break;}
    for (int j=1;j<=N;j++)
        if (use[j]==0) reach[j]=a[t][j],last[j]=t;
    visit[t]=1;
    for (int j=1;j<=N-i;j++)
    {
        maxc=maxk=0;
        for (int k=1;k<=N;k++)
            if ((visit[k]==0) && (reach[k]>maxc)) maxc=reach[k],maxk=k;
        c2=maxk,visit[maxk]=1;
        for (int k=1;k<=N;k++)
            if (visit[k]==0) reach[k]+=a[maxk][k],last[k]=maxk;
    }
    c1=last[c2];
    sum=0;
    for (int j=1;j<=N;j++)
        if (use[j]==0) sum+=a[j][c2];
    ans=min(ans,sum);
    use[c2]=1;
    for (int j=1;j<=N;j++)
        if ((c1!=j) && (use[j]==0)) {a[j][c1]+=a[j][c2];a[c1][j]=a[j][c1];}
}
```

3 Mathematics

3.1 Basics

```
// Greatest Common Divisor & Lowest Common Multiple
ll gcd(ll a, ll b) { return b ? gcd(b, a%b) : a; }
ll lcm(ll a, ll b) { return a*(b/gcd(a, b)); }

// Multiply caring overflow
ll mulmod(ll a, ll b, ll m = MOD) {
    ll r=0;
    for (a %= m; b; b>>=1, a=(a*2)%m) if (b&1) r=(r+a)%m;
    return r;
}

// Another option for mulmod is using long double
ull mulmod(ull a, ull b, ull m = MOD) {
    ull q = (ld) a * (ld) b / (ld) m;
    ull r = a * b - q * m;
    return (r + m) % m;
}

// Fast exponential
ll fexp(ll a, ll b, ll m = MOD) {
    ll r=1;
    for (a %= m; b; b>>=1, a=(a*a)%m) if (b&1) r=(r*a)%m;
    return r;
}

// Multiplicative Inverse
ll inv[N];
inv[1] = 1;
for (int i = 2; i < MOD; ++i)
    inv[i] = (MOD - (MOD/i)*inv[MOD%i]%MOD)%MOD;

// Fibonacci
// Fn = Fn-1 + Fn-2
// F0 = 0 ; F1 = 1
f[0] = 0; f[1] = 1;
for (int i = 2; i <= n; i++) f[i] = f[i-1] + f[i-2];

// Catalan
// Cn = (2n)! / ((n+1)! * n!)
```

```
// 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440
cat[n] = (2*(2*n-1)/(n+1)) * cat[n-1]

// Stirling
// S(n, 1) = S(n, n) = 1
// S(n, k) = k*S(n-1, k) + S(n-1, k-1)
```

3.2 Sieve of Eratosthenes

```
// Sieve of Erasthotenes
int p[N]; vi primes;

for (ll i = 2; i < N; ++i) if (!p[i]) {
    for (ll j = i*i; j < N; j+=i) p[j]=1;
    primes.pb(i);
}
```

3.3 Euler Phi

```
// Euler phi (totient)
int ind = 0, pf = primes[0], ans = n;
while (ll pf*pf <= n) {
    if (n%pf==0) ans -= ans/pf;
    while (n%pf==0) n /= pf;
    pf = primes[++ind];
}
if (n != 1) ans -= ans/n;

// IME2014
int phi[N];
void totient() {
    for (int i = 1; i < N; ++i) phi[i]=i;
    for (int i = 2; i < N; i+=2) phi[i]>>=1;
    for (int j = 3; j < N; j+=2) if (phi[j]==j) {
        phi[j]-=;
        for (int i = 2*j; i < N; i+=j) phi[i]=phi[i]/j*(j-1);
    }
}
```

3.4 Extended Euclidean

```
// Extended Euclid: gcd(a, b) = x*a + y*b
// The solutions are:
// x = x0 + k*b/gcd
// y = y0 + k*a/gcd
void euclid(int a, int b, int &x, int &y, int &d) {
    if (b) euclid(b, a%b, y, x, d), y -= x*(a/b);
    else x = 1, y = 0, d = a;
}
```

3.5 Miller-Rabin

```
// Miller-Rabin - Primarily Test  $O(k*\log n*\log n*\log n)$ 
bool miller(ll a, ll n) {
    if (a >= n) return 1;
    ll s = 0, d = n-1;
    while (d%2 == 0 and d) d >>= 1, s++;
    ll x = fexp(a, d, n);
    if (x == 1 or x == n-1) return 1;
    for (int r = 0; r < s; r++, x = mulmod(x, x, n)) {
        if (x == 1) return 0;
        if (x == n-1) return 1;
    }
    return 0;
}

bool isprime(ll n) {
    int base[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
    for (int i = 0; i < 12; ++i) if (!miller(base[i], n)) return 0;
    return 1;
}
```

```
//n < 2,047 base = {2};
//n < 1,373,653 base = {2, 3};
//n < 9,080,191 base = {31, 73};
//n < 25,326,001 base = {2, 3, 5};
//n < 3,215,031,751 base = {2, 3, 5, 7};
//n < 4,759,123,141 base = {2, 3, 7, 61};
//n < 1,122,004,669,633 base = {2, 13, 23, 1662803};
//n < 2,152,302,898,747 base = {2, 3, 5, 7, 11};
//n < 3,474,749,660,383 base = {2, 3, 5, 7, 11, 13};
//n < 341,550,071,728,321 base = {2, 3, 5, 7, 11, 13, 17};
//n < 3,825,123,056,546,413,051 base = {2, 3, 5, 7, 11, 13, 17, 19, 23};
//n < 318,665,857,834,031,151,167,461 base = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
//n < 3,317,044,064,679,887,385,961,981 base = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41};
```

3.6 Prime Factors

```
// Prime factors (up to  $9*10^{13}$ . For greater see Pollard Rho)
vi factors;
int ind=0, pf = primes[0];
while (pf*pf <= n) {
    while (n%pf == 0) n /= pf, factors.pb(pf);
    pf = primes[++ind];
}
if (n != 1) factors.pb(n);
```

3.7 Pollard-Rho

```
// Pollard Rho - Integer factoring  $O(n^{1/4})$ 
// requires n to be composite (use Miller Rabin to test)
//
ll pollard(ll n) {
    ll x, y, d, c = 1;
    if (n%2==0) return 2;
    while (1) {
        y = x = 2;
        while (1) {
            x = mulmod(x, x, n); x = (x-c)%n;
            y = mulmod(y, y, n); y = (y-c)%n;
            y = mulmod(y, y, n); y = (y-c)%n;
            d = gcd(abs(n+y-x), n);
            if (d == n) break;
            else if (d > 1) return d;
        }
        c++;
    }
}
```

```
// Factorize number using pollar
void fator(ll n, vector<ll>& v) {
    if (isprime(n)) { v.pb(n); return; }

    ll f = pollard(n);
    factorize(f, v); factorize(n/f, v);
}
```

```
// You can optimize the algorithm through the code below
// Using Brent's algorithm for cycle detection
//
std::mt19937 rng((int) std::chrono::steady_clock::now().time_since_epoch().count());

ull func(ull x, ull n, ull c) { return (mulmod(x, x, n) + c) % n; // f(x) = (x^2 + c) % n; }
```

```
ull pollard(ull n) {
    // Finds a positive divisor of n
    ull x, y, d, c;
    ull pot, lam;
    if (n % 2 == 0) return 2;
    if (isprime(n)) return n;

    while(1) {
        y = x = 2; d = 1;
        pot = lam = 1;
        while(1) {
            c = rng() % n;
            if (c != 0 and (c+2)%n != 0) break;
        }
        while(1) {
            if (pot == lam) {
                x = y;
            }
        }
    }
}
```

```

        pot <= 1;
        lam = 0;
    }
    y = func(y, n, c);
    lam++;
    d = gcd(x >= y ? x-y : y-x, n);
    if (d > 1) {
        if (d == n) break;
        else return d;
    }
}
}

void fator(ull n, vector<ull> &v) {
    // prime factorization of n, put into a vector v.
    //
    // for each prime factor of n, it is repeated the amount of times
    // that it divides n
    //
    // ex : n == 120, v = {2, 2, 2, 3, 5};
    //
    if (isprime(n)) { v.pb(n); return; }
    vector<ull> w, t; w.pb(n); t.pb(1);

    while (!w.empty()) {
        ull bck = w.back();
        ull div = pollard(bck);

        if (div == w.back()) {
            int amt = 0;
            for (int i=0; i < (int) w.size(); i++) {
                int cur = 0;
                while (w[i] % div == 0) {
                    w[i] /= div;
                    cur++;
                }
                amt += cur * t[i];
                if (w[i] == 1) {
                    swap(w[i], w.back());
                    swap(t[i], t.back());
                    w.pop_back();
                    t.pop_back();
                }
            }
            while (amt-- > v.pb(div));
        }
        else {
            int amt = 0;
            while (w.back() % div == 0) {
                w.back() /= div;
                amt++;
            }
            amt *= t.back();
            if (w.back() == 1) {
                w.pop_back();
                t.pop_back();
            }

            w.pb(div);
            t.pb(amt);
        }
    }

    // the divisors will not be sorted, so you need to sort it afterwards
    sort(v.begin(), v.end());
}

```

3.8 Fast Fourier Transform

```

// Fast Fourier Transform - O(nlogn)

/*
// Use struct instead. Performance will be way better!
typedef complex<ld> T;
T a[N], b[N];
*/

struct T {
    ld x, y;
    T() : x(0), y(0) {}
    T(ld a, ld b=0) : x(a), y(b) {}

    T operator+=(ld k) { x/=k; y/=k; return (*this); }

```

```

    T operator*(T a) const { return T(x*a.x - y*a.y, x*a.y + y*a.x); }
    T operator+(T a) const { return T(x+a.x, y+a.y); }
    T operator-(T a) const { return T(x-a.x, y-a.y); }
} a[N], b[N];

// a: vector containing polynomial
// n: power of two greater or equal product size
/*
// Use iterative version!
void fft_recursive(T* a, int n, int s) {
    if (n == 1) return;
    T tmp[n];
    for (int i = 0; i < n/2; ++i)
        tmp[i] = a[2*i], tmp[i+n/2] = a[2*i+1];

    fft_recursive(&tmp[0], n/2, s);
    fft_recursive(&tmp[n/2], n/2, s);

    T wn = T(cos(s*2*PI/n), sin(s*2*PI/n)), w(1,0);
    for (int i = 0; i < n/2; i++, w=w*wn)
        a[i] = tmp[i] + w*tmp[i+n/2],
        a[i+n/2] = tmp[i] - w*tmp[i+n/2];
}
*/

void fft(T* a, int n, int s) {
    for (int i=0, j=0; i<n; i++) {
        if (i>j) swap(a[i], a[j]);
        for (int l=n/2; (j^=l) < 1; l>=1);
    }

    for (int i = 1; (1<<i) <= n; i++) {
        int M = 1 << i;
        int K = M >> 1;
        T wn = T(cos(s*2*PI/M), sin(s*2*PI/M));
        for (int j = 0; j < n; j += M) {
            T w = T(1, 0);
            for (int l = j; l < K + j; ++l) {
                T t = w*a[l + K];
                a[l + K] = a[l] - t;
                a[l] = a[l] + t;
                w = wn*w;
            }
        }
    }

    // assert n is a power of two greater of equal product size
    // n = na + nb; while (n%(n-1)) n++;
    void multiply(T* a, T* b, int n) {
        fft(a, n, 1);
        fft(b, n, 1);
        for (int i = 0; i < n; i++) a[i] = a[i]*b[i];
        fft(a, n, -1);
        for (int i = 0; i < n; i++) a[i] /= n;
    }

    // Convert to integers after multiplying:
    // (int) (a[i].x + 0.5);

```

3.9 Fast Fourier Transform(Tourist)

```

//
// FFT made by tourist. It is faster and more supportive, although it requires more lines of code.
// Also, it allows operations with MOD, which is usually an issue in FFT problems.
//
namespace fft {
    typedef double dbl;

    struct num {
        dbl x, y;
        num() { x = y = 0; }
        num(dbl x, dbl y) : x(x), y(y) {}
    };

    inline num operator+ (num a, num b) { return num(a.x + b.x, a.y + b.y); }
    inline num operator- (num a, num b) { return num(a.x - b.x, a.y - b.y); }
    inline num operator* (num a, num b) { return num(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x); }
    inline num conj(num a) { return num(a.x, -a.y); }

    int base = 1;
    vector<num> roots = {{0, 0}, {1, 0}};
    vector<int> rev = {0, 1};

```

```

const dbl PI = acos(-1.0);

void ensure_base(int nbase) {
    if(nbase <= base) return;

    rev.resize(1 << nbase);
    for(int i=0; i < (1 << nbase); i++) {
        rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
    }
    roots.resize(1 << nbase);

    while(base < nbase) {
        dbl angle = 2*PI / (1 << (base + 1));
        for(int i = 1 << (base - 1); i < (1 << base); i++) {
            roots[i << 1] = roots[i];
            dbl angle_i = angle * (2 * i + 1 - (1 << base));
            roots[(i << 1) + 1] = num(cos(angle_i), sin(angle_i));
        }
        base++;
    }
}

void fft(vector<num> &a, int n = -1) {
    if(n == -1) {
        n = a.size();
    }
    assert((n & (n-1)) == 0);
    int zeros = __builtin_ctz(n);
    ensure_base(zeros);
    int shift = base - zeros;
    for(int i = 0; i < n; i++) {
        if(i < (rev[i] >> shift)) {
            swap(a[i], a[rev[i] >> shift]);
        }
    }
    for(int k = 1; k < n; k <= 1) {
        for(int i = 0; i < n; i += 2 * k) {
            for(int j = 0; j < k; j++) {
                num z = a[i+j+k] * roots[j+k];
                a[i+j+k] = a[i+j] - z;
                a[i+j] = a[i+j] + z;
            }
        }
    }
}

vector<num> fa, fb;
vector<int> multiply(vector<int> &a, vector<int> &b) {
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if(sz > (int) fa.size()) {
        fa.resize(sz);
    }
    for(int i = 0; i < sz; i++) {
        int x = (i < (int) a.size() ? a[i] : 0);
        int y = (i < (int) b.size() ? b[i] : 0);
        fa[i] = num(x, y);
    }
    fft(fa, sz);
    num r(0, -0.25 / sz);
    for(int i = 0; i <= (sz >> 1); i++) {
        int j = (sz - i) & (sz - 1);
        num z = (fa[j] * fa[j] - conj(fa[i] * fa[i])) * r;
        if(i != j) {
            fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[j])) * r;
        }
        fa[i] = z;
    }
    fft(fa, sz);
    vector<int> res(need);
    for(int i = 0; i < need; i++) {
        res[i] = fa[i].x + 0.5;
    }
    return res;
}

vector<int> multiply_mod(vector<int> &a, vector<int> &b, int m, int eq = 0) {
    int need = a.size() + b.size() - 1;
    int nbase = 0;
    while((1 << nbase) < need) nbase++;
    ensure_base(nbase);
    int sz = 1 << nbase;
    if (sz > (int) fa.size()) {
        fa.resize(sz);
    }
    for (int i = 0; i < (int) a.size(); i++) {

```

```

        int x = (a[i] % m + m) % m;
        fa[i] = num(x & (1 << 15) - 1), x >> 15);
    }
    fill(fa.begin() + a.size(), fa.begin() + sz, num {0, 0});
    fft(fa, sz);
    if (sz > (int) fb.size()) {
        fb.resize(sz);
    }
    if (eq) {
        copy(fa.begin(), fa.begin() + sz, fb.begin());
    } else {
        for (int i = 0; i < (int) b.size(); i++) {
            int x = (b[i] % m + m) % m;
            fb[i] = num(x & (1 << 15) - 1), x >> 15);
        }
        fill(fb.begin() + b.size(), fb.begin() + sz, num {0, 0});
        fft(fb, sz);
    }
    dbl ratio = 0.25 / sz;
    num r2(0, -1);
    num r3(ratio, 0);
    num r4(0, -ratio);
    num r5(0, 1);
    for (int i = 0; i <= (sz >> 1); i++) {
        int j = (sz - i) & (sz - 1);
        num a1 = (fa[i] + conj(fa[j]));
        num a2 = (fa[i] - conj(fa[j])) * r2;
        num b1 = (fb[i] + conj(fb[j])) * r3;
        num b2 = (fb[i] - conj(fb[j])) * r4;
        if (i != j) {
            num c1 = (fa[j] + conj(fa[i]));
            num c2 = (fa[j] - conj(fa[i])) * r2;
            num d1 = (fb[j] + conj(fb[i])) * r3;
            num d2 = (fb[j] - conj(fb[i])) * r4;
            fa[i] = c1 * d1 + c2 * d2 * r5;
            fb[i] = c1 * d2 + c2 * d1;
        }
        fa[j] = a1 * b1 + a2 * b2 * r5;
        fb[j] = a1 * b2 + a2 * b1;
    }
    fft(fa, sz);
    fft(fb, sz);
    vector<int> res(need);
    for (int i = 0; i < need; i++) {
        long long aa = fa[i].x + 0.5;
        long long bb = fb[i].x + 0.5;
        long long cc = fa[i].y + 0.5;
        res[i] = (aa + ((bb % m) << 15) + ((cc % m) << 30)) % m;
    }
    return res;
}

vector<int> square_mod(vector<int> &a, int m) {
    return multiply_mod(a, a, m, 1);
}
}

```

3.10 Number Theoretic Transform

```
// Number Theoretic Transform - O(nlogn)
```

```
// if long long is not necessary, use int instead to improve performance
const int mod = 20*(1<<23)+1;
const int root = 3;
```

```
ll w[N];
```

```
// a: vector containing polynomial
// n: power of two greater or equal product size
void ntt(ll* a, int n, bool inv) {
    for (int i=0, j=0; i<n; i++) {
        if (i>j) swap(a[i], a[j]);
        for (int l=n/2; (j*=1) < 1; l*=1);
    }
}
```

```
// TODO: Rewrite this loop using FFT version
ll k, t, nrev;
w[0] = 1;
k = exp(root, (mod-1) / n, mod);
for (int i=1; i<n; i++) w[i] = w[i-1] * k % mod;
for (int i=2; i<n; i<=1) for (int j=0; j<n; j+=i) for (int l=0; l<(i/2); l++) {
    int x = j+l, y = j+l+(l/2), z = (n/i)+1;
    t = a[y] * w[inv ? (n-z) : z] % mod;
    a[y] = (a[x] - t + mod) % mod;
    a[x] = (a[j+1] + t) % mod;
}
```



```

}

nrev = exp(n, mod-2, mod);
if (inv) for(int i=0; i<n; ++i) a[i] = a[i] * nrev % mod;
}

// assert n is a power of two greater of equal product size
// n = na + nb; while (n&(n-1)) n++;
void multiply(ll* a, ll* b, int n) {
    ntt(a, n, 0);
    ntt(b, n, 0);
    for (int i = 0; i < n; i++) a[i] = a[i]*b[i] % mod;
    ntt(a, n, 1);
}

```

3.11 Fast Walsh-Hadamard Transform

```

// Fast Walsh-Hadamard Transform - O(nlogn)
//
// Multiply two polynomials, but instead of x^a * x^b = x^(a+b)
// we have x^a * x^b = x^(a XOR b).
//
// WARNING: assert n is a power of two!
void fwht(ll* a, int n, bool inv) {
    for(int l=1; 2*l <= n; l<=1) {
        for(int i=0; i < n; i+=2*l) {
            for(int j=0; j<l; j++) {
                ll u = a[i+j], v = a[i+l+j];

                a[i+j] = (u+v) % MOD;
                a[i+l+j] = (u-v+MOD) % MOD;
                // % is kinda slow, you can use add() macro instead
                // #define add(x,y) (x+y >= MOD ? x+y-MOD : x+y)
            }
        }
    }

    if(inv) {
        for(int i=0; i<n; i++) {
            a[i] = a[i] / n;
        }
    }
}

/* FWHT AND
Matrix : Inverse
0 1   -1 1
1 1     1 0
*/
void fwht_and(vi &a, bool inv) {
    vi ret = a;
    ll u, v;
    int tam = a.size() / 2;
    for(int len = 1; 2 * len <= tam; len <= 1) {
        for(int i = 0; i < tam; i += 2 * len) {
            for(int j = 0; j < len; j++) {
                u = ret[i + j];
                v = ret[i + len + j];
                if(!inv) {
                    ret[i + j] = v;
                    ret[i + len + j] = u + v;
                }
                else {
                    ret[i + j] = -u + v;
                    ret[i + len + j] = u;
                }
            }
        }
    }
    a = ret;
}

```

```

/* FWHT OR
Matrix : Inverse
1 1   0 1
1 0   1 -1
*/
void fwht_or(vi &a, bool inv) {
    vi ret = a;
    ll u, v;
    int tam = a.size() / 2;
    for(int len = 1; 2 * len <= tam; len <= 1) {
        for(int i = 0; i < tam; i += 2 * len) {

```

```

for(int j = 0; j < len; j++) {
    u = ret[i + j];
    v = ret[i + len + j];
    if(!inv) {
        ret[i + j] = u + v;
        ret[i + len + j] = u;
    }
    else {
        ret[i + j] = v;
        ret[i + len + j] = u - v;
    }
}
}
}
a = ret;
}
}

```

3.12 Chinese Remainder

```

// finds z (mod p*q) so z = x (mod p) and z = y (mod q)
ll chinese(ll x, ll p, ll y, ll q) {
    ll s, r, d, mod = p*q;
    euclid(p, q, r, s, d);
    return ((x*(s*q%mod) + y*(r*p%mod) % mod) + mod) % mod;
}

```

3.13 Primitive Root

```

// Finds a primitive root modulo p
ll root(ll p) {
    ll n = p-1;
    vector<ll> fact;

    for (int i=2; i<=n; ++i) if (n % i == 0) {
        fact.push_back (i);
        while (n % i == 0) n /= i;
    }

    if (n > 1) fact.push_back (n);

    for (int res=2; res<=p; ++res) {
        bool ok = true;
        for (size_t i=0; i<fact.size() && ok; ++i)
            ok &= exp(res, (p-1) / fact[i], p) != 1;
        if (ok) return res;
    }

    return -1;
}

```

3.14 Gaussian Elimination (double)

```

//double A[N][M+1], X[M]

for(int j=0; j<m; j++) { //column to eliminate
    int l = j;
    for(int i=j+1; i<n; i++) //find largest pivot
        if(fabs(A[i][j])>fabs(A[l][j]))
            l=i;
    for(int k = 0; k < m+1; k++) { //Swap lines
        swap(A[l][k], A[j][k]);
    }
    for(int i = j+1; i < n; i++) { //eliminate column
        double t=A[i][j]/A[j][j];
        for(int k = j; k < m+1; k++)
            A[i][k]-=t*A[j][k];
    }
}

for(int i = m-1; i >= 0; i--) { //solve triangular system
    for(int j = m-1; j > i; j--)
        A[i][m] -= A[i][j]*X[j];
    X[i]=A[i][m]/A[i][i];
}
}

```

3.15 Gaussian Elimination (modulo prime)

```
//11 A[N][M+1], X[M]

for(int j=0; j<m; j++) { //column to eliminate
    int l = j;
    for(int i=j+1; i<n; i++) //find nonzero pivot
        if(A[i][j]%p)
            l=i;
    for(int k = 0; k < m+1; k++) { //Swap lines
        swap(A[l][k],A[j][k]);
    }
    for(int i = j+1; i < n; i++) { //eliminate column
        ll t=mulmod(A[i][j],inv(A[j][j],p),p);
        for(int k = j; k < m+1; k++)
            A[i][k]=(A[i][k]-mulmod(t,A[j][k],p)+p)%p;
    }
}

for(int i = m-1; i >= 0; i--) { //solve triangular system
    for(int j = m-1; j > i; j--)
        A[i][m] = (A[i][m] - mulmod(A[i][j],X[j],p)+p)%p;
    X[i] = mulmod(A[i][m],inv(A[i][i],p),p);
}
```

3.16 Gaussian Elimination (extended inverse)

```
// Gauss-Jordan Elimination with Scaled Partial Pivoting
// Extended to Calculate Inverses - O(n^3)
// To get more precision choose m[j][i] as pivot the element such that m[j][i] / mx[j] is maximized.
// mx[j] is the element with biggest absolute value of row j.

ld C[N][M]; // N = 1000, M = 2*N+1;
int row, col;

bool elim() {
    for(int i=0; i<row; ++i) {
        int p = i; // Choose the biggest pivot
        for(int j=i; j<row; ++j) if (abs(C[j][i]) > abs(C[p][i])) p = j;
        for(int j=i; j<col; ++j) swap(C[i][j], C[p][j]);

        if (!C[i][i]) return 0;

        ld c = 1/C[i][i]; // Normalize pivot line
        for(int j=0; j<col; ++j) C[i][j] *= c;

        for(int k=i+1; k<col; ++k) {
            ld c = -C[k][i]; // Remove pivot variable from other lines
            for(int j=0; j<col; ++j) C[k][j] += c*C[i][j];
        }
    }

    // Make triangular system a diagonal one
    for(int i=row-1; i>=0; --i) for(int j=i-1; j>=0; --j) {
        ld c = -C[j][i];
        for(int k=i; k<col; ++k) C[j][k] += c*C[i][k];
    }

    return 1;
}

// Finds inv, the inverse of matrix m of size n x n.
// Returns true if procedure was successful.
bool inverse(int n, ld m[N][N], ld inv[N][N]) {
    for(int i=0; i<n; ++i) for(int j=0; j<n; ++j)
        C[i][j] = m[i][j], C[i][j+n] = (i == j);

    row = n, col = 2*n;
    bool ok = elim();

    for(int i=0; i<n; ++i) for(int j=0; j<n; ++j) inv[i][j] = C[i][j+n];
    return ok;
}

// Solves linear system m*x = y, of size n x n
bool linear_system(int n, ld m[N][N], ld *x, ld *y) {
    for(int i=0; i<n; ++i) for(int j=0; j<n; ++j) C[i][j] = m[i][j];
    for(int j=0; j<n; ++j) C[n][j] = x[j];

    row = n, col = n+1;
    bool ok = elim();
}
```

```
for(int j=0; j<n; ++j) y[j] = C[n][j];
return ok;
}
```

3.17 Golden Section Search (Ternary Search)

```
double gss(double l, double r) {
    double m1 = r-(r-l)/gr, m2 = l+(r-l)/gr;
    double f1 = f(m1), f2 = f(m2);
    while(fabs(l-r)>EPS) {
        if(f1>f2) l=m1, f1=f2, m1=m2, m2=l+(r-l)/gr, f2=f(m2);
        else r=m2, f2=f1, m2=m1, m1=r-(r-l)/gr, f1=f(m1);
    }
    return l;
}
```

3.18 Josephus

```
// UFMG
/* Josephus Problem - It returns the position to be, in order to not die. O(n)*/
/* With k=2, for instance, the game begins with 2 being killed and then n+2, n+4, ... */
ll josephus(ll n, ll k) {
    if(n==1) return 1;
    else return (josephus(n-1, k)+k-1)%n+1;
}

/* Another Way to compute the last position to be killed - O(d * log n) */
ll josephus(ll n, ll d) {
    ll K = 1;
    while (K <= (d - 1)*n) K = (d * K + d - 2) / (d - 1);
    return d * n + 1 - K;
}
```

3.19 Simpson Rule

```
// Simpson Integration Rule
// define the function f
double f(double x) {
    // ...
}

double simpson(double a, double b, int n = 1e6) {
    double h = (b - a) / n;
    double s = f(a) + f(b);
    for (int i = 1; i < n; i += 2) s += 4 * f(a + h*i);
    for (int i = 2; i < n; i += 2) s += 2 * f(a + h*i);
    return s*h/3;
}
```

3.20 Discrete Log (Baby-step Giant-step)

```
// solve the discrete equation a^k = m mod(p).
// i.e find k such that the equation above is satisfied.
ll discrete_log(ll a, ll m, ll p) {
    unordered_map<ll, ll> babystep;

    ll b = 1, an = a;
    // set b to the ceil of sqrt(p):
    while(b*b < p) b++, an = (an * a) % p;

    // babysteps:
    ll bstep = m;
    for(ll i=0; i<=b; i++) {
        babystep[bstep] = i;
        bstep = (bstep * a) % p;
    }

    // giantsteps:
    ll gstep = an;
}
```

```

for(ll i=1; i<=b; i++) {
    if(babystep.count(gstep)) {
        return (b * i - babystep[gstep]);
    }
    gstep = (gstep * an) % p;
}

// returns -1 if there isn't any possible value for the answer.
return -1;
}

```

3.21 Mobius Function

```

// 1 if n == 1
// 0 if exists x | n*x == 0
// else (-1)^k, k = #p | p is prime and n*p == 0

int mobius(int n){
    int p = 0;
    int tmp = n;
    for(int i = 2; i*i <= n; i++){
        if(n%i == 0){
            n /= i;
            p++;
            if(n%i == 0) return 0;
        }
    }
    return p%1 ? 1 : -1;
}

```

3.22 Simplex (Stanford)

```

// Two-phase simplex algorithm for solving linear programs of the form
//
//      maximize      c^T x
//      subject to    Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments. Then, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const DOUBLE EPS = 1e-9;

struct LPSolver {
    int m, n;
    VI B, N;
    VVD D;

    LPSolver(const VVD &A, const VD &b, const VD &c) :
        m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
        for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
        for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i]; }
        for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m + 1][n] = 1;
    }

    void Pivot(int r, int s) {
        for (int i = 0; i < m + 2; i++) if (i != r)
            for (int j = 0; j < n + 2; j++) if (j != s)
                D[i][j] -= D[r][j] * D[i][s] / D[r][s];
        for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] /= D[r][s];
        for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] /= -D[r][s];
    }
}

```

```

D[r][s] = 1.0 / D[r][s];
swap(B[r], N[s]);
}

bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m;
    while (true) {
        int s = -1;
        for (int j = 0; j <= n; j++) {
            if (phase == 2 && N[j] == -1) continue;
            if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
        }
        if (D[x][s] > -EPS) return true;
        int r = -1;
        for (int i = 0; i < m; i++) {
            if (D[i][s] < EPS) continue;
            if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
                (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r]) r = i;
        }
        if (r == -1) return false;
        Pivot(r, s);
    }
}

DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -EPS) {
        Pivot(r, n);
        if (!Simplex(1) || D[m + 1][n + 1] < -EPS) return -numeric_limits<DOUBLE>::infinity();
        for (int i = 0; i < m; i++) if (B[i] == -1) {
            int s = -1;
            for (int j = 0; j <= n; j++)
                if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
            Pivot(i, s);
        }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return D[m][n + 1];
}

int main() {
    const int m = 4;
    const int n = 3;
    VVD A(m, VD(n));
    A[0] = { 6, -1, 0 };
    A[1] = { -1, -5, 0 };
    A[2] = { 1, 5, 1 };
    A[3] = { -1, -5, -1 };
    VD b(m);
    b[0] = 10;
    b[1] = -4;
    b[2] = 5;
    b[3] = -5;
    VD c(n);
    c[0] = 1;
    c[1] = -1;
    c[2] = 0;
    LPSolver solver(A, b, c);
    VD x;
    DOUBLE value = solver.Solve(x);

    cerr << "VALUE: " << value << endl; // VALUE: 1.29032
    cerr << "SOLUTION: "; // SOLUTION: 1.74194 0.451613 1
    for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
    cerr << endl;
    return 0;
}

```

4 Strings

4.1 Rabin-Karp

```

// Rabin-Karp - String Matching + Hashing O(n+m)
const int B = 31;
char s[N], p[N];
int n, m; // n = strlen(s), m = strlen(p)

```

```

void rabin() {
    if (n<m) return;

    int hp = 0, hs = 0, E = 1;
    for (int i = 0; i < m; ++i)
        hp = ((hp*B)%MOD + p[i])%MOD,
        hs = ((hs*B)%MOD + s[i])%MOD,
        E = (E*B)%MOD;

    if (hs == hp) { /* matching position 0 */ }
    for (int i = m; i < n; ++i) {
        hs = ((hs*B)%MOD + s[i])%MOD;
        hs = (hs + (MOD - (s[i-m]*E)%MOD)%MOD)%MOD;
        if (hs == hp) { /* matching position i-m+1 */ }
    }
}

```

4.2 Knuth-Morris-Pratt

```

// Knuth-Morris-Pratt - String Matching O(n+m)
char s[N], p[N];
int b[N], n, m; // n = strlen(s), m = strlen(p);

void kmppre() {
    b[0] = -1;
    for (int i = 0, j = -1; i < m; b[++i] = ++j)
        while (j >= 0 and p[i] != p[j])
            j = b[j];
}

void kmp() {
    for (int i = 0, j = 0; i < n; i++) {
        while (j >= 0 and s[i] != p[j]) j=b[j];
        i++, j++;
        if (j == m) {
            /* match position i-j
            j = b[j];
        }
    }
}

```

4.3 Z Function

```

vector<int> z(string s) {
    int n = s.size();
    vector<int> z(n);
    int x = 0, y = 0;
    for (int i = 1; i < n; i++) {
        z[i] = max(0, min(z[i-x], y-i+1));
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
            x = i; y = i+z[i]; z[i]++;
        }
    }
    return z;
}

```

4.4 Prefix Function

```

void p_f(char *s, int *pi) {
    int n = strlen(s);
    for (int i = 2; i <= n; i++) {
        pi[i]=pi[i-1];
        while (pi[i]>0 and s[pi[i]]!=s[i])
            pi[i]=pi[pi[i]];
        if (s[pi[i]]==s[i-1])
            pi[i]++;
    }
}

```

4.5 Recursive-String Matching

```

void p_f(char *s, int *pi) {
    int n = strlen(s);
    pi[0]=pi[1]=0;
    for (int i = 2; i <= n; i++) {
        pi[i] = pi[i-1];
        while (pi[i]>0 and s[pi[i]]!=s[i])
            pi[i]=pi[pi[i]];
        if (s[pi[i]]==s[i-1])
            pi[i]++;
    }
}

int main() {
    //....
    //Initialize prefix function
    char p[N]; //Pattern
    int len = strlen(p); //Pattern size
    int pi[N]; //Prefix function
    p_f(p, pi);

    // Create KMP automaton
    int A[N][128]; //A[i][j]: from state i (size of largest suffix of text which is prefix of pattern),
        append character j -> new state A[i][j]
    for (char c : ALPHABET)
        A[0][c] = (p[0] == c);
    for (int i = 1; p[i]; i++) {
        for (char c : ALPHABET) {
            if (c==p[i])
                A[i][c]=i+1; //match
            else
                A[i][c]=A[pi[i]][c]; //try second largest suffix
        }
    }

    //Create KMP "string appending" automaton
    // g_n = g_(n-1) + char(n) + g_(n-1)
    // g_0 = "", g_1 = "a", g_2 = "aba", g_3 = "abacaba", ...
    int F[M][N]; //F[i][j]: from state j (size of largest suffix of text which is prefix of pattern), append
        string g_i -> new state F[i][j]
    for (int i = 0; i < m; i++) {
        for (int j = 0; j <= len; j++) {
            if (i==0)
                F[i][j] = j; //append empty string
            else {
                int x = F[i-1][j]; //append g_(i-1)
                x = A[x][j]; //append character j
                x = F[i-1][x]; //append g_(i-1)
                F[i][j] = x;
            }
        }
    }

    //Create number of matches matrix
    int K[M][N]; //K[i][j]: from state j (size of largest suffix of text which is prefix of pattern), append
        string g_i -> K[i][j] matches
    for (int i = 0; i < m; i++) {
        for (int j = 0; j <= len; j++) {
            if (i==0)
                K[i][j] = (j==len); //append empty string
            else {
                int x = F[i-1][j]; //append g_(i-1)
                x = A[x][j]; //append character j

                K[i][j] = K[i-1][j] /*append g_(i-1)*/ + (x==len) /*append character j*/ + K[i-1][x]; /*
                    append g_(i-1)*/
            }
        }
    }

    //number of matches in g_k
    int answer = K[0][k];
    //....
}

```

4.6 Aho-Corasick

```

// Aho Corasick - <O(sum(m)), O(n + #matches)>
// Multiple string matching

int p[N], f[N], nxt[N][26], ch[N];
int tsz=1; // size of the trie

int cnt[N]; // used to know number of matches

// used to know which strings matches.
// S is the number of strings. Can use set instead

```

```

const int S = 2e3+5;
bitset<S> elem[N];

void init() {
    tsz=1;
    memset(f, 0, sizeof(f));
    memset(nxt, 0, sizeof(nxt));
    memset(cnt, 0, sizeof(cnt));
    for (int i = 0; i < N; i++) elem[i].reset();
}

void add(const string &s, int x) {
    int cur = 1; // the first element of the trie is the root

    for (int i=0; s[i]; ++i) {
        int j = s[i] - 'a';
        if (!nxt[cur][j]) {
            tsz++;
            p[tsz] = cur;
            ch[tsz] = j;
            nxt[cur][j] = tsz;
        }
        cur = nxt[cur][j];
    }

    cnt[cur]++; //
    elem[cur].set(x);
}

void build() {
    queue<int> q;

    for(int i=0; i<26; ++i) {
        nxt[0][i] = 1;
        if (nxt[1][i]) q.push(nxt[1][i]);
    }

    while (!q.empty()) {
        int v = q.front(); q.pop();
        int u = f[p[v]];

        while (u and !nxt[u][ch[v]]) u = f[u];
        f[v] = nxt[u][ch[v]];
        cnt[v] += cnt[f[v]];

        for (int i = 0; i < 26; ++i) if (nxt[v][i])
            q.push(nxt[v][i]);
    }

    // Return ans to get number of matches
    // Return a map (or global array) if want to know how many of each string have matched
    bitset<S> match(char *s) {
        int ans = 0; // used to know the number of matches
        bitset<S> found; // used to know which strings matches

        int x = 1;
        for (int i = 0; s[i]; ++i) {
            int t = s[i] - 'a';
            while (x and !nxt[x][t]) x = f[x];
            x = nxt[x][t];

            // match found
            ans += cnt[x];
            found |= elem[x];
        }

        return found;
    }
}

```

4.7 Manacher

```

// Manacher (Longest Palindromic String) - O(n)
int lps[2*N+5];
char s[N];

int manacher() {
    int n = strlen(s);

    string p (2*n+3, '#');
    p[0] = '~';
    for (int i = 0; i < n; i++) p[2*(i+1)] = s[i];
    p[2*n+2] = '$';

    int k = 0, r = 0, m = 0;

```

```

int l = p.length();
for (int i = 1; i < l; i++) {
    int o = 2*k - i;
    lps[i] = (r > i) ? min(r-i, lps[o]) : 0;
    while (p[i + 1 + lps[i]] == p[i - 1 - lps[i]]) lps[i]++;
    if (i + lps[i] > r) k = i, r = i + lps[i];
    m = max(m, lps[i]);
}
return m;
}

```

4.8 Suffix Array

```

// Suffix Array O(nlogn)
char s[N];
int n, r, ra[N], sa[N], tra[N], tsa[N], c[N]; // n = strlen(s)

void count(int k) {
    int sum = 0, x = max(300, n); cl(c, 0);
    for (int i = 0; i < n; ++i) c[i+k<n ? ra[i+k] : 0]++;
    for (int i = 0; i < x; ++i) sum += c[i], c[i] = sum-c[i];
    for (int i = 0; i < n; ++i) tsa[c[sa[i]+k<n ? ra[sa[i]+k] : 0]++] = sa[i];
    for (int i = 0; i < n; ++i) sa[i] = tsa[i];
}

void suffixarray() {
    for (int i = 0; i < n; ++i) ra[i] = s[i], sa[i] = i;
    for (int k = 1; k < n; k <= 1) {
        count(k); count(0);
        tra[sa[0]] = r = 0;
        for (int i = 1; i < n; ++i)
            tra[sa[i]] = (ra[sa[i]] == ra[sa[i-1]] and ra[sa[i]+k] == ra[sa[i-1]+k]) ? r : ++r;
        for (int i = 0; i < n; ++i) ra[i] = tra[i];
        if (ra[sa[n-1]] == n-1) break;
    }
}

// String matching with SA O(mlogn)
int mlo, mhi;
int match(char p[], int m) {
    int lo = 0, hi = n-1;
    while (lo < hi) {
        int mid = (lo+hi)/2;
        int res = strncmp(s+sa[mid], p, m);
        if (res < 0) lo = mid+1;
        else hi = mid;
    }
    if (strncmp(s+sa[lo], p, m)) return 0; // no match
    mlo = lo;

    lo = 0, hi = n-1;
    while (lo < hi) {
        int mid = (lo+hi)/2;
        int res = strncmp(s+sa[mid], p, m);
        if (res <= 0) lo = mid+1;
        else hi = mid;
    }
    if (strncmp(s+sa[hi], p, m)) hi--;
    mhi = hi;
    return 1;
}

// Longest Common Prefix with SA O(n)
// strcat(s, "$"); n = strlen(s);
int phi[N], plcp[N], lcp[N];
void calcLCP() {
    int l = 0;
    phi[sa[0]] = -1; plcp[sa[0]] = 0;
    for (int i = 1; i < n; ++i) phi[sa[i]] = sa[i-1];
    for (int i = 0; i < n; ++i) {
        if (phi[i] == -1) continue;
        while (s[i+1] == s[phi[i]+1]) l++;
        plcp[i] = l;
        l = max(l-1, 0);
    }
    for (int i = 0; i < n; ++i) lcp[i] = plcp[sa[i]];
}

// Longest Repeated Substring O(n)
int lrs = 0;
for (int i = 0; i < n; ++i) lrs = max(lrs, lcp[i]);

// Longest Common Substring O(n)
// m = strlen(s);

```

```
// strcat(s, "$"); strcat(s, p); strcat(s, "#");
// n = strlen(s);
int lcs = 0;
for (int i = 1; i < n; ++i) if ((sa[i] < m) != (sa[i-1] < m))
    lcs = max(lcs, lcp[i]);
```

4.9 Suffix Automaton

```
// Suffix Automaton Construction - O(n)

const int N = 1e6+1, K = 26;
int sl[2*N], len[2*N], sz, last;
ll cnt[2*N];
map<int, int> adj[2*N];

void add(int c) {
    int u = sz++;
    len[u] = len[last] + 1;
    cnt[u] = 1;

    int p = last;
    while(p != -1 and !adj[p][c])
        adj[p][c] = u, p = sl[p];

    if (p == -1) sl[u] = 0;
    else {
        int q = adj[p][c];
        if (len[p] + 1 == len[q]) sl[u] = q;
        else {
            int r = sz++;
            len[r] = len[p] + 1;
            sl[r] = sl[q];
            adj[r] = adj[q];
            while(p != -1 and adj[p][c] == q)
                adj[p][c] = r, p = sl[p];
            sl[q] = sl[u] = r;
        }
    }

    last = u;
}

void clear() {
    for(int i=0; i<=sz; ++i) adj[i].clear();
    last = 0;
    sz = 1;
    sl[0] = -1;
}

void build(char *s) {
    clear();
    for(int i=0; s[i]; ++i) add(s[i]);
}

// Pattern matching - O(|p|)
bool check(char *p) {
    int u = 0, ok = 1;
    for(int i=0; p[i]; ++i) {
        u = adj[u][p[i]];
        if (!u) ok = 0;
    }
    return ok;
}

// Substring count - O(|p|)
ll d[2*N];

void substr_cnt(int u) {
    d[u] = 1;
    for(auto p : adj[u]) {
        int v = p.second;
        if (!d[v]) substr_cnt(v);
        d[u] += d[v];
    }
}

ll substr_cnt() {
    memset(d, 0, sizeof d);
    substr_cnt(0);
    return d[0] - 1;
}

// k-th Substring - O(|s|)
// Just find the k-th path in the automaton.
// Can be done with the value d calculated in previous problem.
```

```
// Smallest cyclic shift - O(|s|)
// Build the automaton for string s + s. And adapt previous dp
// to only count paths with size |s|.

// Number of occurrences - O(|p|)
vector<int> t[2*N];

void occur_count(int u) {
    for(int v : t[u]) occur_count(v), cnt[u] += cnt[v];
}

void build_tree() {
    for(int i=1; i<=sz; ++i)
        t[sl[i]].push_back(i);
    occur_count(0);
}

ll occur_count(char *p) {
    // Call build tree once per automaton
    int u = 0;
    for(int i=0; p[i]; ++i) {
        u = adj[u][p[i]];
        if (!u) break;
    }
    return !u ? 0 : cnt[u];
}

// First occurrence - (|p|)
// Store the first position of occurrence fp.
// Add the the code to add function:
// fp[u] = len[u] - 1;
// fp[r] = fp[q];

// To answer a query, just output fp[u] - strlen(p) + 1
// where u is the state corresponding to string p

// All occurrences - O(|p| + |ans|)
// All the occurrences can reach the first occurrence via suffix links.
// So every state that contains a occurrence is reachable by the
// first occurrence state in the suffix link tree. Just do a DFS in this
// tree, starting from the first occurrence.
// OBS: cloned nodes will output same answer twice.

// Smallest substring not contained in the string - O(|s| * K)
// Just do a dynamic programming:
// d[u] = 1 // if d does not have 1 transition
// d[u] = 1 + min d[v] // otherwise

// LCS of 2 Strings - O(|s| + |t|)
// Build automaton of s and traverse the automaton with string t
// maintaining the current state and the current length.
// When we have a transition: update state, increase length by one.
// If we don't update state by suffix link and the new length will
// should be reduced (if bigger) to the new state length.
// Answer will be the maximum length of the whole traversal.

// LCS of n Strings - O(n*|s|*K)
// Create a new string S = s_1 + d_1 + ... + s_n + d_n,
// where d_i are delimiters that are unique (d_i != d_j).
// For each state use DP + bitmask to calculate if it can
// reach a d_i transition without going through other d_j.
// The answer will be the biggest len[u] that can reach all
// d_i's.
```

4.10 Z Function (Stanford)

```
void compute_z_function(const char *S, int N) {
    // Z[i] is the length of longest substring
    // starting from S[i] which matches a prefix of S.
    int L = 0, R = 0;
    for (int i = 1; i < N; ++i) {
        if (i > R) {
            L = R = i;
            while (R < N && S[R - L] == S[R]) ++R;
            Z[i] = R - L; --R;
        } else {
            int k = i - L;
            if (Z[k] < R - i + 1) Z[i] = Z[k];
            else {
                L = i;
                while (R < N && S[R - L] == S[R]) ++R;
                Z[i] = R - L; --R;
            }
        }
    }
}
```

```

        while (R < N && S[R - k] == S[R]) ++R;
        Z[i] = R - L; --R;
    }
}
}
}

```

5 Data Structures

5.1 Disjoint Set Union

```

// Disjoint Set Union / Union-find
int par[N], sz[N];

int find(int a) { return par[a] == a ? a : par[a] = find(par[a]); }

void unite(int a, int b) {
    if ((a = find(a)) == (b = find(b))) return;
    if (sz[a] < sz[b]) swap(a, b);
    par[b] = a; sz[a] += sz[b];
}

```

5.2 Sparse Table

```

struct Sparse{
    int n, t[30][N];
    void init(int _n, int *v) {
        n = _n; f = _f;
        for(int i = 0; i < n; i++)
            t[0][i] = v[i];
        for(int i = 0; i < 29; i++)
            for(int j = 0; j + (1<<i) < n; j++)
                t[i+1][j] = min(t[i][j], t[i][j + (1<<i)]);
    }
    int qry(int l, int r) {
        int r = INF;
        for(int i = 29; l != r; i--)
            if(l + (1<<i) <= r)
                r = min(r, t[i][l]);
        return r = min(r, t[0][r]);
    }
};

```

5.3 Sparse Table 2D

```

// 2D Sparse Table - <O(n^2 (log n) ^ 2), O(1)>
const int N = 1e3+1, M = 10;
int t[N][N], v[N][N], dp[M][M][N][N], lg[N], n, m;

void build() {
    int k = 0;
    for(int i=1; i<N; ++i) {
        if (1<=k == i/2) k++;
        lg[i] = k;
    }

    // Set base cases
    for(int x=0; x<n; ++x) for(int y=0; y<m; ++y) dp[0][0][x][y] = v[x][y];
    for(int j=1; j<M; ++j) for(int x=0; x<n; ++x) for(int y=0; y+(1<<j)<=m; ++y)
        dp[0][j][x][y] = max(dp[0][j-1][x][y], dp[0][j-1][x][y+(1<<j-1)]);

    // Calculate sparse table values
    for(int i=1; i<M; ++i) for(int j=0; j<M; ++j)
        for(int x=0; x+(1<<i)<=n; ++x) for(int y=0; y+(1<<j)<=m; ++y)
            dp[i][j][x][y] = max(dp[i-1][j][x][y], dp[i-1][j][x+(1<<i-1)][y]);
}

int query(int x1, int x2, int y1, int y2) {
    int i = lg[x2-x1+1], j = lg[y2-y1+1];
    int m1 = max(dp[i][j][x1][y1], dp[i][j][x2-(1<<i)+1][y1]);
    int m2 = max(dp[i][j][x1][y2-(1<<j)+1], dp[i][j][x2-(1<<i)+1][y2-(1<<j)+1]);
    return max(m1, m2);
}

```

5.4 Fenwick Tree

```

// Fenwick Tree / Binary Indexed Tree
int bit[N];

void add(int p, int v) {
    for (p+=2; p<N; p+=p&-p) bit[p] += v;
}

int query(int p) {
    int r = 0;
    for (p+=2; p; p-=p&-p) r += bit[p];
    return r;
}

```

5.5 Fenwick Tree 2D

```

// Fenwick Tree 2D / Binary Indexed Tree 2D
int bit[N][N];

void add(int i, int j, int v) {
    for (; i < N; i+=i&-1)
        for (int jj = j; jj < N; jj+=jj&-jj)
            bit[i][jj] += v;
}

int query(int i, int j) {
    int res = 0;
    for (; i; i-=i&-1)
        for (int jj = j; jj; jj-=jj&-jj)
            res += bit[i][jj];
    return res;
}

// Whole BIT 2D set to 1
void init() {
    cl(bit, 0);
    for (int i = 1; i <= r; ++i)
        for (int j = 1; j <= c; ++j)
            add(i, j, 1);
}

// Return number of positions set
int query(int imin, int jmin, int imax, int jmax) {
    return query(imax, jmax) - query(imax, jmin-1) - query(imin-1, jmax) + query(imin-1, jmin-1);
}

// Find all positions inside rect (imin, jmin), (imax, jmax) where position is set
void proc(int imin, int jmin, int imax, int jmax, int v, int tot) {
    if (tot < 0) tot = query(imin, jmin, imax, jmax);
    if (!tot) return;

    int imid = (imin+imax)/2, jmid = (jmin+jmax)/2;
    if (imin != imax) {
        int qnt = query(imin, jmin, imid, jmax);
        if (qnt) proc(imin, jmin, imid, jmax, v, qnt);
        if (tot-qnt) proc(imid+1, jmin, imax, jmax, v, tot-qnt);
    } else if (jmin != jmax) {
        int qnt = query(imin, jmin, imax, jmid);
        if (qnt) proc(imin, jmin, imax, jmid, v, qnt);
        if (tot-qnt) proc(imin, jmid+1, imax, jmax, v, tot-qnt);
    } else {
        // single position set!
        // now process position!!!
        add(imin, jmin, -1);
    }
}

```

5.6 Range Update Point Query Fenwick Tree

```

struct BIT {
    ll b[N]={};
    ll sum(int x) {
        ll r=0;
        for(x+=2;x;x-=x&-x)
            r += b[x];
        return r;
    }
};

```

```

    }
    void upd(int x, ll v) {
        for(x+=2; x<N; x+=x&-x)
            b[x]+=v;
    }
};

struct BITRange {
    BIT a,b;
    ll sum(int x) {
        return a.sum(x)*x+b.sum(x);
    }
    void upd(int l, int r, ll v) {
        a.upd(l,v), a.upd(r+1,-v);
        b.upd(l,-v*(l-1)), b.upd(r+1,v*r);
    }
};

```

5.7 Segment Tree

```

// Segment Tree
int st[4*N], v[N], lz[4*N];

void build(int p, int l, int r) {
    if (l == r) { st[p] = v[l]; return; }
    build(2*p, l, (l+r)/2);
    build(2*p+1, (l+r)/2+1, r);
    st[p] = min(st[2*p], st[2*p+1]); // RMQ -> min/max, RSQ -> +
}

void push(int p, int l, int r) {
    if (lz[p]) {
        st[p] = lz[p];
        // RMQ -> update: = lz[p], increment: += lz[p]
        // RSQ -> update: = (r-l+1)*lz[p], increment: += (r-l+1)*lz[p]
        if(l!=r) lz[2*p] = lz[2*p+1] = lz[p]; // update: =, increment +=
        lz[p] = 0;
    }
}

int query(int p, int l, int r, int i, int j) {
    push(p, l, r);
    if (r < i or l > j) return INF; // RMQ -> INF, RSQ -> 0
    if (l >= i and r <= j) return st[p];
    return min(query(2*p, l, (l+r)/2, i, j),
               query(2*p+1, (l+r)/2+1, r, i, j));
    // RMQ -> min/max, RSQ -> +
}

void update(int p, int l, int r, int i, int j, int v) {
    push(p, l, r);
    if (r < i or l > j) return;
    if (l >= i and r <= j) { lz[p] = v; push(p, l, r); return; }
    update(2*p, l, (l+r)/2, i, j, v);
    update(2*p+1, (l+r)/2+1, r, i, j, v);
    st[p] = min(st[2*p], st[2*p+1]); // RMQ -> min/max, RSQ -> +
}

```

5.8 Segment Tree 2D

```

// Segment Tree 2D - O(nlog(n)log(n)) of Memory and Runtime
const int N = 1e8+5, M = 2e5+5;
int n, k=1, st[N], lc[N], rc[N];

void addx(int x, int l, int r, int u) {
    if (x < l or r < x) return;

    st[u]++;
    if (l == r) return;

    if(!rc[u]) rc[u] = ++k, lc[u] = ++k;
    addx(x, l, (l+r)/2, lc[u]);
    addx(x, (l+r)/2+1, r, rc[u]);
}

// Adds a point (x, y) to the grid.
void add(int x, int y, int l, int r, int u) {
    if (y < l or r < y) return;

    if (!st[u]) st[u] = ++k;
    addx(x, l, n, st[u]);
}

```

```

    if (l == r) return;

    if(!rc[u]) rc[u] = ++k, lc[u] = ++k;
    add(x, y, l, (l+r)/2, lc[u]);
    add(x, y, (l+r)/2+1, r, rc[u]);
}

int countx(int x, int l, int r, int u) {
    if (!u or x < l) return 0;
    if (r <= x) return st[u];

    return countx(x, l, (l+r)/2, lc[u]) +
           countx(x, (l+r)/2+1, r, rc[u]);
}

// Counts number of points dominated by (x, y)
// Should be called with l = 1, r = n and u = 1
int count(int x, int y, int l, int r, int u) {
    if (!u or y < l) return 0;
    if (r <= y) return countx(x, l, n, st[u]);

    return count(x, y, l, (l+r)/2, lc[u]) +
           count(x, y, (l+r)/2+1, r, rc[u]);
}

```

5.9 Persistent Segment Tree

```

// Persistent Segtree
// Memory: O(n logn)
// Operations: O(log n)

int li[N], ri[N]; // [li(u), ri(u)] is the interval of node u
int st[N], lc[N], rc[N]; // Value, left son and right son of node u
int stsz; // Size of segment tree

// Returns root of initial tree.
// i and j are the first and last elements of the tree.
int init(int i, int j) {
    int v = ++stsz;
    li[v] = i, ri[v] = j;

    if (i != j) {
        rc[v] = init(i, (i+j)/2);
        rc[v] = init((i+j)/2+1, j);
        st[v] = /* calculate value from rc[v] and rc[v] */;
    } else {
        st[v] = /* insert initial value here */;
    }

    return v;
}

// Gets the sum from i to j from tree with root u
int sum(int u, int i, int j) {
    if (j < li[u] or ri[u] < i) return 0;
    if (i <= li[u] and ri[u] <= j) return st[u];
    return sum(rc[u], i, j) + sum(rc[u], i, j);
}

// Copies node j into node i
void clone(int i, int j) {
    li[i] = li[j], ri[i] = ri[j];
    st[i] = st[j];
    rc[i] = rc[j], rc[i] = rc[j];
}

// Sums v to index i from the tree with root u
int update(int u, int i, int v) {
    if (i < li[u] or ri[u] < i) return u;

    clone(++stsz, u);
    u = stsz;
    rc[u] = update(rc[u], i, v);
    rc[u] = update(rc[u], i, v);

    if (li[u] == ri[u]) st[u] += v;
    else st[u] = st[rc[u]] + st[rc[u]];

    return u;
}

```


5.10 Persistent Segment Tree (Naum)

```
// Persistent Segment Tree
int n;
int rcnt;
int lc[M], rc[M], st[M];

int update(int p, int l, int r, int i, int v) {
    int rt = ++rcnt;
    if (l == r) { st[rt] = v; return rt; }

    int mid = (l+r)/2;
    if (i <= mid) lc[rt] = update(lc[p], l, mid, i, v), rc[rt] = rc[p];
    else rc[rt] = update(rc[p], mid+1, r, i, v), lc[rt] = lc[p];
    st[rt] = st[lc[rt]] + st[rc[rt]];

    return rt;
}

int query(int p, int l, int r, int i, int j) {
    if (l > j or r < i) return 0;
    if (l <= i and r <= j) return st[p];

    return query(lc[p], l, (l+r)/2, i, j) + query(rc[p], (l+r)/2+1, r, i, j);
}

int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i) {
        int a;
        scanf("%d", &a);
        r[i] = update(r[i-1], 1, n, i, 1);
    }

    return 0;
}
```

5.11 Heavy-Light Decomposition

```
// Heavy-Light Decomposition
vector<int> adj[N];
int par[N], h[N];

int chainno, chain[N], head[N], chainpos[N], chainsz[N], pos[N], arrsz;
int sc[N], sz[N];

void dfs(int u) {
    sz[u] = 1, sc[u] = 0; // nodes 1-indexed (0-ind: sc[u]==-1)
    for (int v : adj[u]) if (v != par[u]) {
        par[v] = u, h[v] = h[u]+1, dfs(v);
        sz[u] += sz[v];
        if (sz[sc[u]] < sz[v]) sc[u] = v; // 1-indexed (0-ind: sc[u]<0 or ...)
    }
}

void hld(int u) {
    if (!head[chainno]) head[chainno] = u; // 1-indexed
    chain[u] = chainno;
    chainpos[u] = chainsz[chainno];
    chainsz[chainno]++;
    pos[u] = ++arrsz;

    if (sc[u]) hld(sc[u]);

    for (int v : adj[u]) if (v != par[u] and v != sc[u])
        chainno++, hld(v);
}

int lca(int u, int v) {
    while (chain[u] != chain[v]) {
        if (h[head[chain[u]]] < h[head[chain[v]]]) swap(u, v);
        u = par[head[chain[u]]];
    }
    if (h[u] > h[v]) swap(u, v);
    return u;
}

int query_up(int u, int v) {
    if (u == v) return 0;
    int ans = -1;
    while (1) {

```

```
        if (chain[u] == chain[v]) {
            if (u == v) break;
            ans = max(ans, query(1, 1, n, chainpos[v]+1, chainpos[u]));
            break;
        }

        ans = max(ans, query(1, 1, n, chainpos[head[chain[u]]], chainpos[u]));
        u = par[head[chain[u]]];
    }
    return ans;
}

int query(int u, int v) {
    int l = lca(u, v);
    return max(query_up(u, l), query_up(v, l));
}
```

5.12 Heavy-Light Decomposition (new)

```
const int N = 2e5 + 5;
int par[N];
vector<int> adj[N];
int sz[N], nxt[N];
int h[N];
int in[N], rin[N], out[N];
int t;

void dfs_sz(int u = 1) {
    sz[u] = 1;
    for (auto &v : adj[u]) {
        h[v] = h[u] + 1;

        dfs_sz(v);
        sz[u] += sz[v];
        if (sz[v] > sz[adj[u][0]])
            swap(v, adj[u][0]);
    }
}

void dfs_hld(int u = 1) {
    in[u] = t++;
    rin[in[u]] = u;
    for (auto v : adj[u]) {
        nxt[v] = (v == adj[u][0] ? nxt[u] : v);
        dfs_hld(v);
    }

    out[u] = t - 1;
}

int lca(int u, int v) {
    while (nxt[u] != nxt[v]) {
        if (h[nxt[u]] < h[nxt[v]]) swap(u, v);
        u = par[nxt[u]];
    }

    if (h[u] > h[v]) swap(u, v);
    return u;
}
```

5.13 Centroid Decomposition

```
// Centroid decomposition
vector<int> adj[N];
int forb[N], sz[N], par[N];
int n, m;
unordered_map<int, int> dist[N];

void dfs(int u, int p) {
    sz[u] = 1;
    for (int v : adj[u]) {
        if (v != p and !forb[v]) {
            dfs(v, u);
            sz[u] += sz[v];
        }
    }
}

int find_cen(int u, int p, int qt) {

```

```

for(int v : adj[u]) {
    if(v == p or forb[v]) continue;
    if(sz[v] > qt / 2) return find_cen(v, u, qt);
}
return u;
}

void getdist(int u, int p, int cen) {
    for(int v : adj[u]) {
        if(v != p and !forb[v]) {
            dist[cen][v] = dist[v][cen] = dist[cen][u] + 1;
            getdist(v, u, cen);
        }
    }
}

void decomp(int u, int p) {
    dfs(u, -1);

    int cen = find_cen(u, -1, sz[u]);
    forb[cen] = 1;
    par[cen] = p;
    dist[cen][cen] = 0;
    getdist(cen, -1, cen);

    for(int v : adj[cen]) if(!forb[v])
        decomp(v, cen);
}

// main
decomp(1, -1);

```

5.14 Trie

```

// Trie <O(|S|), O(|S|)>
int trie[N][26], trien = 1;

int add(int u, char c){
    c-='a';
    if (trie[u][c]) return trie[u][c];
    return trie[u][c] = ++trien;
}

//to add a string s in the trie
int u = 1;
for(char c : s) u = add(u, c);

```

5.15 Mergesort Tree

```

// Mergesort Tree - Time <O(nlognlogn), O(nlogn)> - Memory O(nlogn)
// Mergesort Tree is a segment tree that stores the sorted subarray
// on each node.
vi st[4*N];

void build(int p, int l, int r) {
    if (l == r) { st[p].pb(s[l]); return; }
    build(2*p, l, (l+r)/2);
    build(2*p+1, (l+r)/2+1, r);
    st[p].resize(r-l+1);
    merge(st[2*p].begin(), st[2*p].end(),
          st[2*p+1].begin(), st[2*p+1].end(),
          st[p].begin());
}

int query(int p, int l, int r, int i, int j, int a, int b) {
    if (j < l or i > r) return 0;
    if (i <= l and j >= r)
        return upper_bound(st[p].begin(), st[p].end(), b) -
               lower_bound(st[p].begin(), st[p].end(), a);
    return query(2*p, l, (l+r)/2, i, j, a, b) +
           query(2*p+1, (l+r)/2+1, r, i, j, a, b);
}

```

5.16 Treap

```

// Treap (probabilistic BST)
// O(logn) operations (supports lazy propagation)

mt19937_64 llrand(random_device{}());

struct node {
    int val;
    int cnt, rev;
    int mn, mx, mindiff; // value-based treap only!
    ll pri;
    node* l;
    node* r;

    node() {}
    node(int x) : val(x), cnt(1), rev(0), mn(x), mx(x), mindiff(INF), pri(llrand()), l(0), r(0) {}
};

struct treap {
    node* root;
    treap() : root(0) {}
    ~treap() { clear(); }

    int cnt(node* t) { return t ? t->cnt : 0; }
    int mn (node* t) { return t ? t->mn : INF; }
    int mx (node* t) { return t ? t->mx : -INF; }
    int mindiff(node* t) { return t ? t->mindiff : INF; }

    void clear() { del(root); }
    void del(node* t) {
        if (!t) return;
        del(t->l); del(t->r);
        delete t;
        t = 0;
    }

    void push(node* t) {
        if (!t or !t->rev) return;
        swap(t->l, t->r);
        if (t->l) t->l->rev ^= 1;
        if (t->r) t->r->rev ^= 1;
        t->rev = 0;
    }

    void update(node*& t) {
        if (!t) return;
        t->cnt = cnt(t->l) + cnt(t->r) + 1;
        t->mn = min(t->val, min(mn(t->l), mn(t->r)));
        t->mx = max(t->val, max(mx(t->l), mx(t->r)));
        t->mindiff = min(mn(t->r) - t->val, min(t->val - mx(t->l), min(mindiff(t->l), mindiff(t->r))));
    }

    node* merge(node* l, node* r) {
        push(l); push(r);
        node* t;
        if (!l or !r) t = l ? l : r;
        else if (l->pri > r->pri) l->r = merge(l->r, r), t = l;
        else r->l = merge(l, r->l), t = r;
        update(t);
        return t;
    }

    pair<node*, node*> split(node* t, int pos) {
        if (!t) return {0, 0};
        push(t);

        if (cnt(t->l) < pos) {
            auto x = split(t->r, pos-cnt(t->l)-1);
            t->r = x.st;
            update(t);
            return { t, x.nd };
        }

        auto x = split(t->l, pos);
        t->l = x.nd;
        update(t);
        return { x.st, t };
    }

    // Position-based treap
    // used when the values are just additional data
    // the positions are known when it's built, after that you
    // query to get the values at specific positions
    /*
    void insert(int pos, int val) {
        push(root);
        node* x = new node(val);
        auto t = split(root, pos);
        root = merge(merge(t.st, x), t.nd);
    }
    */
}

```

```

void erase(int pos) {
    auto t1 = split(root, pos);
    auto t2 = split(t1.nd, 1);
    delete t2.st;
    root = merge(t1.st, t2.nd);
}

int get_val(int pos) { return get_val(root, pos); }
int get_val(node* t, int pos) {
    push(t);
    if (cnt(t->l) == pos) return t->val;
    if (cnt(t->l) < pos) return get_val(t->r, pos-cnt(t->l)-1);
    return get_val(t->l, pos);
}
*/
// -----
// Value-based treap
// used when the values needs to be ordered
int order(node* t, int val) {
    if (!t) return 0;
    push(t);
    if (t->val < val) return cnt(t->l) + 1 + order(t->r, val);
    return order(t->l, val);
}

bool has(node* t, int val) {
    if (!t) return 0;
    push(t);
    if (t->val == val) return 1;
    return has((t->val > val ? t->l : t->r), val);
}

void insert(int val) {
    if (has(root, val)) return; // avoid repeated values
    push(root);
    node* x = new node(c);
    auto t = split(root, order(root, val));
    root = merge(merge(t.st, x), t.nd);
}

void erase(int val) {
    if (!has(root, val)) return;

    auto t1 = split(root, order(root, val));
    auto t2 = split(t1.nd, 1);
    delete t2.st;
    root = merge(t1.st, t2.nd);
}

// Get the maximum difference between values
int querymax(int i, int j) {
    if (i == j) return -1;
    auto t1 = split(root, j+1);
    auto t2 = split(t1.st, i);

    int ans = mx(t2.nd) - mn(t2.nd);
    root = merge(merge(t2.st, t2.nd), t1.nd);
    return ans;
}

// Get the minimum difference between values
int querymin(int i, int j) {
    if (i == j) return -1;
    auto t2 = split(root, j+1);
    auto t1 = split(t2.st, i);

    int ans = mindiff(t1.nd);
    root = merge(merge(t1.st, t1.nd), t2.nd);
    return ans;
}
// -----

void reverse(int l, int r) {
    auto t2 = split(root, r+1);
    auto t1 = split(t2.st, l);
    t1.nd->rev = 1;
    root = merge(merge(t1.st, t1.nd), t2.nd);
}

void print() { print(root); printf("\n"); }
void print(node* t) {
    if (!t) return;
    push(t);
    print(t->l);
    printf("%d ", t->val);
    print(t->r);
}

```

};

5.17 KD Tree (Stanford)

```

const int maxn=200005;

struct kdtree
{
    int xl,xr,yl,yr,zl,zr,max,flag; // flag=0:x axis 1:y 2:z
} tree[5000005];

int N,M,lastans,xq,yq;
int a[maxn],pre[maxn],nxt[maxn];
int x[maxn],y[maxn],z[maxn],wei[maxn];
int xc[maxn],yc[maxn],zc[maxn],wc[maxn],hash[maxn],biao[maxn];

bool cmp1(int a,int b)
{
    return x[a]<x[b];
}

bool cmp2(int a,int b)
{
    return y[a]<y[b];
}

bool cmp3(int a,int b)
{
    return z[a]<z[b];
}

void makekdtree(int node,int l,int r,int flag)
{
    if (l>r)
    {
        tree[node].max=-maxlongint;
        return;
    }
    int xl=maxlongint,xr=-maxlongint;
    int yl=maxlongint,yr=-maxlongint;
    int zl=maxlongint,zr=-maxlongint,maxc=-maxlongint;
    for (int i=l;i<=r;i++)
        xl=min(xl,x[i]),xr=max(xr,x[i]),
        yl=min(yl,y[i]),yr=max(yr,y[i]),
        zl=min(zl,z[i]),zr=max(zr,z[i]),
        maxc=max(maxc,wei[i]),
        xc[i]=x[i],yc[i]=y[i],zc[i]=z[i],wc[i]=wei[i],biao[i]=i;
    tree[node].flag=flag;
    tree[node].xl=xl,tree[node].xr=xr,tree[node].yl=yl;
    tree[node].yr=yr,tree[node].zl=zl,tree[node].zr=zr;
    tree[node].max=maxc;
    if (l==r) return;
    if (flag==0) sort(biao+l,biao+r+1,cmp1);
    if (flag==1) sort(biao+l,biao+r+1,cmp2);
    if (flag==2) sort(biao+l,biao+r+1,cmp3);
    for (int i=l;i<=r;i++)
        x[i]=xc[biao[i]],y[i]=yc[biao[i]],
        z[i]=zc[biao[i]],wei[i]=wc[biao[i]];
    makekdtree(node+2,l,(l+r)/2,(flag+1)%3);
    makekdtree(node+2+1,(l+r)/2+1,r,(flag+1)%3);
}

int getmax(int node,int xl,int xr,int yl,int yr,int zl,int zr)
{
    xl=max(xl,tree[node].xl);
    xr=min(xr,tree[node].xr);
    yl=max(yl,tree[node].yl);
    yr=min(yr,tree[node].yr);
    zl=max(zl,tree[node].zl);
    zr=min(zr,tree[node].zr);
    if (tree[node].max==maxlongint) return 0;
    if ((xr<tree[node].xl)|| (xl>tree[node].xr)) return 0;
    if ((yr<tree[node].yl)|| (yl>tree[node].yr)) return 0;
    if ((zr<tree[node].zl)|| (zl>tree[node].zr)) return 0;
    if ((tree[node].xl==xl)&&(tree[node].xr==xr)&&
        (tree[node].yl==yl)&&(tree[node].yr==yr)&&
        (tree[node].zl==zl)&&(tree[node].zr==zr))
        return tree[node].max;
    else
        return max(getmax(node+2,xl,xr,yl,yr,zl,zr),
            getmax(node+2+1,xl,xr,yl,yr,zl,zr));
}

int main()

```

```
{
    // N 3D-rect with weights
    // find the maximum weight containing the given 3D-point
    return 0;
}
```

5.18 Maximum Queue

```
// O(1) complexity for all operations, except for clear,
// which could be done by creating another deque and using swap

struct MaxQueue {
    int plus = 0;
    deque<pair<int, int>> dq;

    bool empty() { return dq.size() == 0; }
    void clear() { plus = 0; dq.clear(); }
    void add(int x) { plus += x; } // Adds x to every element in the queue
    int max() { return dq.begin()->first + plus; } // Returns the maximum element in the queue

    void push(int x) {
        x -= plus;
        int amt = 0;
        while (dq.size() and dq.back().first <= x)
            amt += dq.back().second + 1, dq.pop_back();
        dq.push_back({ x, amt });
    }

    void pop() {
        if (dq.empty()) return;
        if (!dq.front().second) dq.pop_front();
        else dq.front().second--;
    }
};
```

5.19 Ordered Set

```
//#include <ext/pb_ds/assoc_container.hpp>
//#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/detail/standard_policies.hpp>
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;

ordered_set s;
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);

//find_by_order returns an iterator to the element at a given position
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7

//order_of_key returns the position of a given element
cout << s.order_of_key(7) << "\n"; // 2

//If the element does not appear in the set, we get the position that the element would have in the set
cout << s.order_of_key(6) << "\n"; // 2
cout << s.order_of_key(8) << "\n"; // 3
```

5.20 Lichao Tree (ITA)

```
#include <cstdio>
#include <vector>
#define INF 0x3f3f3f3f3f3f3f3f
#define MAXN 1009
using namespace std;

typedef long long ll;

/*
 * LiChao Segment Tree
 */
```

```
class LiChao {
    vector<ll> m, b;
    int n, sz; ll *x;
#define gx(i) (i < sz ? x[i] : x[sz-1])
    void update(int t, int l, int r, ll nm, ll nb) {
        ll xl = nm * gx(l) + nb, xr = nm * gx(r) + nb;
        ll yl = m[t] * gx(l) + b[t], yr = m[t] * gx(r) + b[t];
        if (yl >= xl && yr >= xr) return;
        if (yl <= xl && yr <= xr) {
            m[t] = nm, b[t] = nb; return;
        }
        int mid = (l + r) / 2;
        update(t<l, l, mid, nm, nb);
        update(1+(t<l), mid+1, r, nm, nb);
    }
public:
    LiChao(ll *st, ll *en) : x(st) {
        sz = int(en - st);
        for(n = 1; n < sz; n <= 1);
        m.assign(2*n, 0); b.assign(2*n, -INF);
    }
    void insert_line(ll nm, ll nb) {
        update(1, 0, n-1, nm, nb);
    }
    ll query(int i) {
        ll ans = -INF;
        for(int t = i+n; t; t >= 1)
            ans = max(ans, m[t] * x[i] + b[t]);
        return ans;
    }
};

/*
 * UVa 12524
 */

ll w[MAXN], x[MAXN], A[MAXN], B[MAXN], dp[MAXN][MAXN];

int main(){
    int N, K;
    while(scanf("%d %d", &N, &K)!=EOF) {
        for(int i=0; i<N; i++){
            scanf("%lld %lld", x+i, w+i);
            A[i] = w[i] + (i>0 ? A[i-1] : 0);
            B[i] = w[i]*x[i] + (i>0 ? B[i-1] : 0);
            dp[i][1] = x[i]*A[i] - B[i];
        }
        for(int k=2; k<=K; k++){
            dp[0][k] = 0;
            LiChao lc(x, x+N);
            for(int i=1; i<N; i++){
                lc.insert_line(A[i-1], -dp[i-1][k-1]-B[i-1]);
                dp[i][k] = x[i]*A[i] - B[i] - lc.query(i);
            }
            printf("%lld\n", dp[N-1][K]);
        }
        return 0;
    }
}
```

6 Dynamic Programming

6.1 Longest Increasing Subsequence

```
// Longest Increasing Subsequence - O(nlogn)
//
// dp(i) = max j<i { dp(j) | a[j] < a[i] } + 1
//
int dp[N], v[N], n, lis;

memset(dp, 63, sizeof dp);
for (int i=0; i<n; ++i) {
    // increasing: lower_bound
    // non-decreasing: upper_bound
    int j = lower_bound(dp, dp+lis, v[i]) - dp;
    dp[j] = min(dp[j], v[i]);
    lis = max(lis, j+1);
}
```

6.2 Convex Hull Trick

```
// Convex Hull Trick

// In case of floating point parameters swap long long with long double
typedef long long type;
struct line { type b, m; }

// Order by increasing m if b is equal, otherwise by decreasing b
// If you need the minimum convex hull use { -b, -m }
// [](line s, line t){ return s.b == t.b ? s.m < t.m : s.b > t.b }

// nh: number of lines on convex hull
// pos: position for linear time search
// hull: lines in the convex hull
int nh, pos;
line hull[N];

bool check(line s, line t, line u) {
    return (s.b - t.b)*(u.m - s.m) < (s.b - u.b)*(t.m - s.m);
}

// Add new line to convex hull, if possible
// Must receive lines in the correct order, otherwise it won't work
void update(line s) {
    // 1. if first lines have the same b, get the one with bigger m
    // 2. if line is parallel to the one at the top, ignore
    // 3. pop lines that are worse
    // 3.1 if you can do a linear time search, use
    // 4. add new line

    if (nh == 1 and hull[nh-1].b == s.b) nh--;
    if (nh > 0 and hull[nh-1].m >= s.m) return;
    while (nh >= 2 and !check(hull[nh-2], hull[nh-1], s)) nh--;
    hull[nh++] = s;
}

type eval(int id, type x) { return hull[id].b + hull[id].m * x; }

/*
// Linear search query - O(n) for all queries
// Only possible if the queries always move to the right
type query(type x) {
    while (pos+1 < nh and eval(pos, x) > eval(pos+1, x)) pos++;
    return eval(pos, x);
}
*/

// Ternary search query - O(logn) for each query
type query(type x) {
    int lo = 0, hi = nh-1;
    while (lo < hi) {
        int mid = (lo+hi)/2;
        if (eval(mid, x) < eval(mid+1, x)) hi = mid;
        else lo = mid+1;
    }
    return eval(lo, x);
}

// better use geometry line_intersect (this assumes s and t are not parallel)
ld intersect_x(line s, line t) { return (t.b - s.b)/(ld)(s.m - t.m); }
ld intersect_y(line s, line t) { return s.b + s.m * intersect_x(s, t); }
```

6.3 Convex Hull Trick (emaxx)

```
struct Point{
    ll x, y;
    Point(ll x = 0, ll y = 0):x(x), y(y) {}
    Point operator-(Point p){ return Point(x - p.x, y - p.y); }
    Point operator+(Point p){ return Point(x + p.x, y + p.y); }
    Point ccw(){ return Point(-y, x); }
    ll operator%(Point p){ return x*p.y - y*p.x; }
    ll operator*(Point p){ return x*p.x + y*p.y; }
    bool operator<(Point p) const { return x == p.x ? y < p.y : x < p.x; }
};

pair<vector<Point>, vector<Point>> ch(Point *v){
    vector<Point> hull, vecs;
    for(int i = 0; i < n; i++){
        if(hull.size() and hull.back().x == v[i].x) continue;
```

```
while(vecs.size() and vecs.back().x(v[i] - hull.back()) <= 0)
    vecs.pop_back(), hull.pop_back();

if(hull.size())
    vecs.pb(v[i] - hull.back());

hull.pb(v[i]);
}
return {hull, vecs};
}

ll get(ll x) {
    Point query = {x, 1};
    auto it = lower_bound(vecs.begin(), vecs.end(), query, [](Point a, Point b) {
        return a%b > 0;
    });
    return query*hull[it - vecs.begin()];
}
```

6.4 Divide and Conquer Optimization

```
// Divide and Conquer DP Optimization - O(k*n^2) => O(k*n*logn)
//
// dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
// Condition: A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[i][j]
// reference (pt-br): https://algorithmmarch.wordpress.com/2016/08/12/a-otimizacao-de-pds-e-o-garcom-da-maratona/

int n, maxj;
int dp[N][J], a[N][J];

// declare the cost function
int cost(int i, int j) {
    // ...
}

void calc(int l, int r, int j, int kmin, int kmax) {
    int m = (l+r)/2;
    dp[m][j] = LINF;

    for (int k = kmin; k <= kmax; ++k) {
        ll v = dp[k][j-1] + cost(k, m);

        // store the minimum answer for d[m][j]
        // in case of maximum, use v > dp[m][j]
        if (v < dp[m][j]) a[m][j] = k, dp[m][j] = v;
    }

    if (l < r) {
        calc(l, m, j, kmin, a[m][k]);
        calc(m+1, r, j, a[m][k], kmax);
    }
}

// run for every j
for (int j = 2; j <= maxj; ++j)
    calc(1, n, j, 1, n);
```

6.5 Knuth Optimization

```
// Knuth DP Optimization - O(n^3) -> O(n^2)
//
// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
//
// Condition: A[i][j-1] <= A[i][j] <= A[i+1][j]
// A[i][j] is the smallest k that gives an optimal answer to dp[i][j]
// reference (pt-br): https://algorithmmarch.wordpress.com/2016/08/12/a-otimizacao-de-pds-e-o-garcom-da-maratona/
//
// 1) dp[i][j] = min i<k<j { dp[i][k] + dp[k][j] } + C[i][j]
int n;
int dp[N][N], a[N][N];

// declare the cost function
```

```

int cost(int i, int j) {
    // ...
}

void knuth() {
    // calculate base cases
    memset(dp, 63, sizeof(dp));
    for (int i = 1; i <= n; i++) dp[i][i] = 0;

    // set initial a[i][j]
    for (int i = 1; i <= n; i++) a[i][i] = i;

    for (int j = 2; j <= n; ++j)
        for (int i = j; i >= 1; --i)
            for (int k = a[i][j-1]; k <= a[i+1][j]; ++k) {
                ll v = dp[i][k] + dp[k][j] + cost(i, j);

                // store the minimum answer for d[i][k]
                // in case of maximum, use v > dp[i][k]
                if (v < dp[i][j])
                    a[i][j] = k, dp[i][j] = v;
            }
}

// 2) dp[i][j] = min k<i { dp[k][j-1] + C[k][i] }
int n, maxj;
int dp[N][J], a[N][J];

// declare the cost function
int cost(int i, int j) {
    // ...
}

void knuth() {
    // calculate base cases
    memset(dp, 63, sizeof(dp));
    for (int i = 1; i <= n; i++) dp[i][1] = // ...

    // set initial a[i][j]
    for (int i = 1; i <= n; i++) a[i][0] = 0, a[n+1][i] = n;

    for (int j = 2; j <= maxj; j++)
        for (int i = n; i >= 1; i--)
            for (int k = a[i][j-1]; k <= a[i+1][j]; k++) {
                ll v = dp[k][j-1] + cost(k, i);

                // store the minimum answer for d[i][k]
                // in case of maximum, use v > dp[i][k]
                if (v < dp[i][j])
                    a[i][j] = k, dp[i][j] = v;
            }
}

```

7 Geometry

7.1 Basic

```

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

const int INF = 0x3f3f3f3f;

typedef long double ld;
const double EPS = 1e-9, PI = acos(-1.);

// Change long double to long long if using integers
typedef long double type;

bool ge(type x, type y) { return x + EPS > y; }
bool le(type x, type y) { return x - EPS < y; }
bool eq(type x, type y) { return ge(x, y) and le(x, y); }

struct point {
    type x, y;

    point() : x(0), y(0) {}
    point(type x, type y) : x(x), y(y) {}

    point operator -( ) { return point(-x, -y); }
    point operator +(point p) { return point(x+p.x, y+p.y); }
    point operator -(point p) { return point(x-p.x, y-p.y); }
}

```

```

point operator *(type k) { return point(k*x, k*y); }
point operator /(type k) { return point(x/k, y/k); }

type operator *(point p) { return x*p.x + y*p.y; }
type operator %(point p) { return x*p.y - y*p.x; }

// o is the origin, p is another point
// dir == +1 => p is clockwise from this
// dir == 0 => p is colinear with this
// dir == -1 => p is counterclockwise from this
int dir(point o, point p) {
    type x = (*this - o) % (p - o);
    return ge(x, 0) - le(x, 0);
}

bool on_seg(point p, point q) {
    if (this->dir(p, q) return 0;
    return ge(x, min(p.x, q.x)) and le(x, max(p.x, q.x)) and
        ge(y, min(p.y, q.y)) and le(y, max(p.y, q.y));
}

ld abs() { return sqrt(x*x + y*y); }
type abs2() { return x*x + y*y; }
ld dist(point x) { return (*this - x).abs(); }
type dist2(point x) { return (*this - x).abs2(); }

ld arg() { return atan2l(y, x); }

// Project point on vector y
point project(point y) { return y * ((*this * y) / (y * y)); }

// Project point on line generated by points x and y
point project(point x, point y) { return x + (*this - x).project(y-x); }

ld dist_line(point x, point y) { return dist(project(x, y)); }

ld dist_seg(point x, point y) {
    return project(x, y).on_seg(x, y) ? dist_line(x, y) : min(dist(x), dist(y));
}

point rotate(ld sin, ld cos) { return point(cos*x-sin*y, sin*x+cos*y); }
point rotate(ld a) { return rotate(sin(a), cos(a)); }
// rotate around the argument of vector p
point rotate(point p) { return rotate(p.x / p.abs(), p.y / p.abs()); }
};

int direction(point o, point p, point q) { return p.dir(o, q); }

bool segments_intersect(point p, point q, point a, point b) {
    int d1, d2, d3, d4;
    d1 = direction(p, q, a);
    d2 = direction(p, q, b);
    d3 = direction(a, b, p);
    d4 = direction(a, b, q);
    if (d1*d2 < 0 and d3*d4 < 0) return 1;
    return p.on_seg(a, b) or q.on_seg(a, b) or
        a.on_seg(p, q) or b.on_seg(p, q);
}

point lines_intersect(point p, point q, point a, point b) {
    point r = q-p, s = b-a, c(p%q, a%b);
    if (eq(r%s, 0)) return point(INF, INF);
    return point(point(r.x, s.x) % c, point(r.y, s.y) % c) / (r%s);
}

// Sorting points in counterclockwise order.
// If the angle is the same, closer points to the origin come first.
point origin;
bool radial(point p, point q) {
    int dir = p.dir(origin, q);
    return dir > 0 or (!dir and p.on_seg(origin, q));
}

// Graham Scan
vector<point> convex_hull(vector<point> pts) {
    vector<point> ch(pts.size());
    point mn = pts[0];

    for(point p : pts) if (p.y < mn.y or (p.y == mn.y and p.x < p.y)) mn = p;

    origin = mn;
    sort(pts.begin(), pts.end(), radial);

    int n = 0;

    // IF: Convex hull without collinear points
    for(point p : pts) {

```

```

    while (n > 1 and ch[n-1].dir(ch[n-2], p) < 1) n--;
    ch[n++] = p;
}

/* ELSE IF: Convex hull with collinear points
for(point p : pts) {
    while (n > 1 and ch[n-1].dir(ch[n-2], p) < 0) n--;
    ch[n++] = p;
}

for(int i=pts.size()-1; i >=1; --i)
    if (pts[i] != ch[n-1] and !pts[i].dir(pts[0], ch[n-1]))
        ch[n++] = pts[i];
// END IF */

ch.resize(n);
return ch;
}

// Double of the triangle area
ld double_of_triangle_area(point p1, point p2, point p3) {
    return abs((p2-p1) % (p3-p1));
}

// TODO: test this code. This code has not been tested, please do it before proper use.
// http://codeforces.com/problemset/problem/975/E is a good problem for testing.
point centroid(vector<point> &v) {
    int n = v.size();
    type da = 0;
    point m, c;

    for(point p : v) m = m + p;
    m = m / n;

    for(int i=0; i<n; ++i) {
        point p = v[i] - m, q = v[(i+1)%n] - m;
        type x = p % q;
        c = c + (p + q) * x;
        da += x;
    }

    return c / (3 * da);
}

bool point_inside_triangle(point p, point p1, point p2, point p3) {
    ld a1, a2, a3, a;
    a = double_of_triangle_area(p1, p2, p3);
    a1 = double_of_triangle_area(p, p2, p3);
    a2 = double_of_triangle_area(p, p1, p3);
    a3 = double_of_triangle_area(p, p1, p2);
    return eq(a, a1 + a2 + a3);
}

bool point_inside_convex_poly(int l, int r, vector<point> v, point p) {
    while (l+1 != r) {
        int m = (l+r)/2;
        if (p.dir(v[0], v[m])) r = m;
        else l = m;
    }
    return point_inside_triangle(p, v[0], v[l], v[r]);
}

vector<point> circle_circle_intersection(point p1, ld r1, point p2, ld r2) {
    vector<point> ret;

    ld d = p1.dist(p2);
    if (d > r1 + r2 or d + min(r1, r2) < max(r1, r2)) return ret;

    ld x = (r1*r1 - r2*r2 + d*d) / (2*d);
    ld y = sqrt(r1*r1 - x*x);

    point v = (p2 - p1)/d;

    ret.push_back(p1 + v * x + v.rotate(PI/2) * y);
    if (y > 0)
        ret.push_back(p1 + v * x - v.rotate(PI/2) * y);

    return ret;
}

```

```

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

```

```

const int INF = 0x3f3f3f3f;

typedef long double ld;
const double EPS = 1e-9, PI = acos(-1.);

// Change long double to long long if using integers
typedef long double type;

//return -1: a < b, 0: a == b, 1: a > b
int comp(type a, type b) {
    return (a > b + EPS) - (a < b - EPS);
}

struct point {
    type x, y;

    point(type x = 0, type y = 0) : x(x), y(y) {}

    point operator -() { return point(-x, -y); }
    point operator +(point p) { return point(x+p.x, y+p.y); }
    point operator -(point p) { return point(x-p.x, y-p.y); }

    point operator *(type k) { return point(k*x, k*y); }
    point operator /(type k) { return point(x/k, y/k); }

    type operator *(point p) { return x*p.x + y*p.y; }
    type operator %(point p) { return x*p.y - y*p.x; }
    type operator !() {return (*this)*(*this); };

    bool onSegment(point a, point b) {
        if(comp((*this-a)%(b-a), 0)) return 0;
        return (comp(x, min(a.x, b.x)) >= 0 and
                comp(x, max(a.x, b.x)) <= 0 and
                comp(y, min(a.y, b.y)) >= 0 and
                comp(y, max(a.y, b.y)) <= 0);
    }
};

ostream &operator<<(ostream &os, const point &p) {
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}

point rotateCCW(point p, ld t) {
    return point(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

point projectPointOnLine(point p, point a, point b) {
    return a + (b-a)*((p-a)*(b-a))/((b-a)*(b-a));
}

point projectPointSegment(point p, point a, point b) {
    ld r = (b-a)*(b-a);
    if(abs(r) < EPS) return a;
    r = ((p-a)*(b-a))/r;
    if(r < 0) return a;
    if(r > 1) return b;
    return a + (b-a)*r;
}

ld distPointSegment(point p, point a, point b) {
    return sqrt(1-(projectPointSegment(p, a, b) - p));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
ld distPointPlane(ld x, ld y, ld z,
                  ld a, ld b, ld c, ld d) {
    return abs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

bool linesCollinear(point a, point b, point c, point d) {
    return abs((a-b)%(c-d)) < EPS and
           abs((a-b)%(a-c)) < EPS;
}

bool segmentIntersect(point a, point b, point c, point d) {
    if(linesCollinear(a, b, c, d)) {
        if (! (a - c) < EPS or ! (a - d) < EPS or
            ! (b - c) < EPS or ! (b - d) < EPS) return true;
        if ((c-a)*(c-b) > 0 && (d-a)*(d-b) > 0 && (c-b)*(d-b) > 0)
            return false;
        return true;
    }
    //overflow
    if (((d-a)%(b-a)) * ((c-a)%(b-a)) > 0) return false;
    if (((a-c)%(d-c)) * ((b-c)%(d-c)) > 0) return false;
    return true;
}

```

7.2 Basic (new)

```

point lineIntersection(point a, point b, point c, point d) {
    b = b-a; d = c-d; c = c-a;
    return a + b*(c%d)/(b%d);
}

point circleCenter(point a, point b, point c) {
    b=(a+b)/2;
    c=(a+c)/2;
    return lineIntersection(b, b+rotateCCW(a-b, -PI/2),
                           c, c+rotateCCW(a-c, -PI/2));
}

// Sorting points in counterclockwise order.
// If the angle is the same, closer points to the origin come first.
point origin;
bool radial(point a, point b) {
    double cp = (a - origin)%(b - origin);
    return abs(cp) < EPS ? !(a - origin) < !(b - origin) : cp > 0;
}

// Graham Scan
vector<point> convex_hull(vector<point> &pts) {
    vector<point> ch(pts.size());
    point mn = pts[0];

    for(point p : pts) if (p.y < mn.y or (p.y == mn.y and p.x < p.y)) mn = p;

    origin = mn;
    sort(pts.begin(), pts.end(), radial);

    int n = 0;

    // IF: Convex hull without collinear points
    for(point p : pts) {
        while (n > 1 and (ch[n - 1] - ch[n - 2])%(p - ch[n - 2]) < EPS) n--;
        ch[n++] = p;
    }

    /* ELSE IF: Convex hull with collinear points
    for(point p : pts) {
        while (n > 1 and (ch[n - 1] - ch[n - 2])%(p - ch[n - 2]) < -EPS) n--;
        ch[n++] = p;
    }

    for(int i=pts.size()-1; i >=1; --i)
        if (pts[i] != ch[n-1] and !pts[i].dir(pts[0], ch[n-1]))
            ch[n++] = pts[i];
    // END IF */

    ch.resize(n);
    return ch;
}

//Shoe Lace
double signedArea(vector<point> &p){
    double area = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        area += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return area / 2.0;
}

// TODO: test this code. This code has not been tested, please do it before proper use.
// http://codeforces.com/problemset/problem/975/E is a good problem for testing.
point centroid(vector<point> &p) {
    point c;
    double scale = 6.0 * signedArea(p);
    for (int i = 0; i < p.size(); i++){
        int j = (i+1) % p.size();
        c = c + (p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

// Random on border
bool pointInPolygon(const vector<point> &p, point q) {
    bool c = 0;
    for (int i = 0; i < p.size(); i++){
        int j = (i+1) % p.size();
        if ((p[i].y <= q.y && q.y < p[j].y ||
            p[j].y <= q.y && q.y < p[i].y) &&
            q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
            c = !c;
    }
    return c;
}

```

```

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<point> circleLineIntersection(point a, point b, point c, double r) {
    vector<point> ret;
    b = b-a;
    a = a-c;
    double A = b*b;
    double B = a*b;
    double C = a*a - r*r;
    double D = B*B - A*C;
    if (D < -EPS) return ret;
    ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
    if (D > EPS)
        ret.push_back(c+a+b*(-B-sqrt(D))/A);
    return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<point> circleCircleIntersection(point c1, point c2, double r1, double r2) {
    vector<point> ret;
    double d = sqrt((c1 - c2).x*(c1 - c2).x);
    if (d > r1+r2 || d+min(r1, r2) < max(r1, r2)) return ret;
    double x = (d*d-r2*r2+r1*r1)/(2*d);
    double y = sqrt(r1*r1-x*x);
    point v = (c2-c1)/d;
    ret.push_back(c1 + v*x + rotateCCW(v, PI/2)*y);
    if (y > 0)
        ret.push_back(c1 + v*x - rotateCCW(v, PI/2)*y);
    return ret;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool isSimple(const vector<point> &p) {
    for (int i = 0; i < p.size(); i++) {
        for (int k = i+1; k < p.size(); k++) {
            int j = (i+1) % p.size();
            int l = (k+1) % p.size();
            if (i == l || j == k) continue;
            if (segmentIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

int main() {
    return 0;
}

```

7.3 Closest Pair of Points

//Time complexity: $O(n \log n)$, using merge sort strategy

```

struct pnt{
    long long x, y;
    pnt operator-(pnt p){ return {x - p.x, y - p.y}; }
    long long operator!(){ return x*x+y*y; }
};

const int N = 1e5 + 5;
pnt pnts[N];
pnt tmp[N];
pnt p1, p2;
unsigned long long d = 9e18;

void closest(int l, int r){
    if(l == r) return;
    int mid = (l + r)/2;

    int midx = pnts[mid].x;
    closest(l, mid), closest(mid + 1, r);

    merge(pnts + l, pnts + mid + 1, pnts + mid + 1, pnts + r + 1, tmp + l,
          [](pnt a, pnt b){ return a.y < b.y; });

    for (int i = l; i <= r; i++) pnts[i] = tmp[i];

    vector<point> margin;
    for(int i = l; i <= r; i++){
        if((pnts[i].x - midx)*(pnts[i].x - midx) < d)
            margin.push_back(pnts[i]);
    }

    for(int i = 0; i < margin.size(); i++)
        for(int j = i + 1;

```



```

        j < margin.size() and
        (margin[j].y - margin[i].y)*(margin[j].y - margin[i].y) < d;
    j++) {
        if(!(margin[i] - margin[j]) < d)
            p1 = margin[i], p2 = margin[j], d = !(p1 - p2);
    }
}

```

7.4 Nearest Neighbours

```

// Closest Neighbor - O(n * log(n))
const ll N = 1e6+3, INF = 1e18;
ll n, cn[N], x[N], y[N]; // number of points, closes neighbor, x coordinates, y coordinates

ll sqr(ll i) { return i*i; }
ll dist(int i, int j) { return sqr(x[i]-x[j]) + sqr(y[i]-y[j]); }
ll dist(int i) { return i == cn[i] ? INF : dist(i, cn[i]); }

bool cpx(int i, int j) { return x[i] < x[j] or (x[i] == x[j] and y[i] < y[j]); }
bool cpy(int i, int j) { return y[i] < y[j] or (y[i] == y[j] and x[i] < x[j]); }

ll calc(int i, ll x0) {
    ll dlt = dist(i) - sqr(x[i]-x0);
    return dlt >= 0 ? ceil(sqrt(dlt)) : -1;
}

void updt(int i, int j, ll x0, ll &dlt) {
    if (dist(i) > dist(i, j)) cn[i] = j, dlt = calc(i, x0);
}

void cmp(vi &u, vi &v, ll x0) {
    for(int a=0, b=0; a<u.size(); ++a) {
        ll i = u[a], dlt = calc(i, x0);
        while(b < v.size() and y[i] > y[v[b]]) b++;
        for(int j = b-1; j >= 0 and y[i] - dlt <= y[v[j]]; j--) updt(i, v[j], x0, dlt);
        for(int j = b; j < v.size() and y[i] + dlt >= y[v[j]]; j++) updt(i, v[j], x0, dlt);
    }
}

void slv(vi &ix, vi &iy) {
    int n = ix.size();
    if (n == 1) { cn[ix[0]] = ix[0]; return; }

    int m = ix[n/2];

    vi ix1, ix2, iy1, iy2;
    for(int i=0; i<n; ++i) {
        if (cpx(ix[i], m)) ix1.push_back(ix[i]);
        else ix2.push_back(ix[i]);

        if (cpy(iy[i], m)) iy1.push_back(iy[i]);
        else iy2.push_back(iy[i]);
    }

    slv(ix1, iy1);
    slv(ix2, iy2);

    cmp(iy1, iy2, x[m]);
    cmp(iy2, iy1, x[m]);
}

void slv(int n) {
    vi ix, iy;
    ix.resize(n);
    iy.resize(n);
    for(int i=0; i<n; ++i) ix[i] = iy[i] = i;
    sort(ix.begin(), ix.end(), cpx);
    sort(iy.begin(), iy.end(), cpy);
    slv(ix, iy);
}

```

8 Geometry (Stanford)

8.1 Basic

```

// C++ routines for computational geometry.
double INF = 1e100;

```

```

double EPS = 1e-12;

struct PT {
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c); }
    PT operator / (double c) const { return PT(x/c, y/c); }
};

double dot(PT p, PT q) { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p) { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
    return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
    return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
    double r = dot(b-a,b-a);
    if (fabs(r) < EPS) return a;
    r = dot(c-a, b-a)/r;
    if (r < 0) return a;
    if (r > 1) return b;
    return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
    double a, double b, double c, double d)
{
    return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
    return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a, b, c, d)
        && fabs(cross(a-b, a-c)) < EPS
        && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if (LinesCollinear(a, b, c, d)) {
        if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
            dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
        if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
            return false;
        return true;
    }
    if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
    if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
    return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
    b=b-a; d=d-c; c=c-a;
    assert(dot(b, b) > EPS && dot(d, d) > EPS);
}

```

```

    return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
    b=(a+b)/2;
    c=(a+c)/2;
    return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
    bool c = 0;
    for (int i = 0; i < p.size(); i++){
        int j = (i+1)%p.size();
        if ((p[i].y <= q.y && q.y < p[j].y ||
            p[j].y <= q.y && q.y < p[i].y) &&
            q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
            c = !c;
    }
    return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
    for (int i = 0; i < p.size(); i++)
        if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
            return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
    vector<PT> ret;
    b = b-a;
    a = a-c;
    double A = dot(b, b);
    double B = dot(a, b);
    double C = dot(a, a) - r*r;
    double D = B*B - A*C;
    if (D < -EPS) return ret;
    ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
    if (D > EPS)
        ret.push_back(c+a+b*(-B-sqrt(D))/A);
    return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
    vector<PT> ret;
    double d = sqrt(dist2(a, b));
    if (d > r+R || d+min(r, R) < max(r, R)) return ret;
    double x = (d*d-R*R+r*r)/(2*d);
    double y = sqrt(r*r-x*x);
    PT v = (b-a)/d;
    ret.push_back(a+v*x + RotateCCW90(v)*y);
    if (y > 0)
        ret.push_back(a+v*x - RotateCCW90(v)*y);
    return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion. Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
    double area = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        area += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
    return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
    PT c(0,0);
    double scale = 6.0 * ComputeSignedArea(p);

```

```

    for (int i = 0; i < p.size(); i++){
        int j = (i+1) % p.size();
        c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
    for (int i = 0; i < p.size(); i++) {
        for (int k = i+1; k < p.size(); k++) {
            int j = (i+1) % p.size();
            int l = (k+1) % p.size();
            if (i == 1 || j == k) continue;
            if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

int main() {

    // expected: (-5,2)
    cerr << RotateCCW90(PT(2,5)) << endl;

    // expected: (5,-2)
    cerr << RotateCW90(PT(2,5)) << endl;

    // expected: (-5,2)
    cerr << RotateCCW(PT(2,5), M_PI/2) << endl;

    // expected: (5,2)
    cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

    // expected: (5,2) (7.5,3) (2.5,1)
    cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
        << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
        << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

    // expected: 6.78903
    cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

    // expected: 1 0 1
    cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
        << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
        << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

    // expected: 0 0 1
    cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
        << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
        << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

    // expected: 1 1 1 0
    cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

    // expected: (1,2)
    cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;

    // expected: (1,1)
    cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

    vector<PT> v;
    v.push_back(PT(0,0));
    v.push_back(PT(5,0));
    v.push_back(PT(5,5));
    v.push_back(PT(0,5));

    // expected: 1 1 1 0 0
    cerr << PointInPolygon(v, PT(2,2)) << " "
        << PointInPolygon(v, PT(2,0)) << " "
        << PointInPolygon(v, PT(0,2)) << " "
        << PointInPolygon(v, PT(5,2)) << " "
        << PointInPolygon(v, PT(2,5)) << endl;

    // expected: 0 1 1 1 1
    cerr << PointOnPolygon(v, PT(2,2)) << " "
        << PointOnPolygon(v, PT(2,0)) << " "
        << PointOnPolygon(v, PT(0,2)) << " "
        << PointOnPolygon(v, PT(5,2)) << " "
        << PointOnPolygon(v, PT(2,5)) << endl;

    // expected: (1,6)
    //          (5,4) (4,5)
    //          blank line
    //          (4,5) (5,4)

```

```
//      blank line
//      (4,5) (5,4)
vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

// area should be 5.0
// centroid should be (1.1666666, 1.1666666)
PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
vector<PT> p(pa, pa+4);
PT c = ComputeCentroid(p);
cerr << "Area: " << ComputeArea(p) << endl;
cerr << "Centroid: " << c << endl;

return 0;
}
```

8.2 Convex Hull

```
// Compute the 2D convex hull of a set of points using the monotone chain
// algorithm. Eliminate redundant points from the hull if REMOVE_REDUNDANT is
// defined.
//
// Running time: O(n log n)
//
// INPUT:   a vector of input points, unordered.
// OUTPUT:  a vector of points in the convex hull, counterclockwise, starting
//          with bottommost/leftmost point

#include <stdio>
#include <cassert>
#include <vector>
#include <algorithm>
#include <cmath>
// BEGIN CUT
#include <map>
// END CUT

using namespace std;

#define REMOVE_REDUNDANT

typedef double T;
const T EPS = 1e-7;
struct PT {
    T x, y;
    PT() {}
    PT(T x, T y) : x(x), y(y) {}
    bool operator<(const PT &rhs) const { return make_pair(y,x) < make_pair(rhs.y,rhs.x); }
    bool operator==(const PT &rhs) const { return make_pair(y,x) == make_pair(rhs.y,rhs.x); }
};

T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
    return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
    sort(pts.begin(), pts.end());
    pts.erase(unique(pts.begin(), pts.end(), pts.end()), pts.end());
    vector<PT> up, dn;
    for (int i = 0; i < pts.size(); i++) {
        while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
        while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
        up.push_back(pts[i]);
        dn.push_back(pts[i]);
    }
    pts = dn;
    for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
    if (pts.size() <= 2) return;
#endif
}
```

```
dn.clear();
dn.push_back(pts[0]);
dn.push_back(pts[1]);
for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
    dn.push_back(pts[i]);
}
if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
}
pts = dn;
#endif
}

// BEGIN CUT
// The following code solves SPOJ problem #26: Build the Fence (BSHEEP)

int main() {
    int t;
    scanf("%d", &t);
    for (int caseno = 0; caseno < t; caseno++) {
        int n;
        scanf("%d", &n);
        vector<PT> v(n);
        for (int i = 0; i < n; i++) scanf("%lf%lf", &v[i].x, &v[i].y);
        vector<PT> h(v);
        map<PT,int> index;
        for (int i = n-1; i >= 0; i--) index[v[i]] = i+1;
        ConvexHull(h);

        double len = 0;
        for (int i = 0; i < h.size(); i++) {
            double dx = h[i].x - h[(i+1)%h.size()].x;
            double dy = h[i].y - h[(i+1)%h.size()].y;
            len += sqrt(dx*dx+dy*dy);
        }

        if (caseno > 0) printf("\n");
        printf("%.2f\n", len);
        for (int i = 0; i < h.size(); i++) {
            if (i > 0) printf(" ");
            printf("%d", index[h[i]]);
        }
        printf("\n");
    }
}

// END CUT
```

8.3 Delaunay Triangulation

```
// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:   x[] = x-coordinates
//          y[] = y-coordinates
//
// OUTPUT:  triples = a vector containing m triples of indices
//          corresponding to triangle vertices

#include<vector>
using namespace std;

typedef double T;

struct triple {
    int i, j, k;
    triple() {}
    triple(int i, int j, int k) : i(i), j(j), k(k) {}
};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
    int n = x.size();
    vector<T> z(n);
    vector<triple> ret;

    for (int i = 0; i < n; i++)
        z[i] = x[i] * x[i] + y[i] * y[i];

    for (int i = 0; i < n-2; i++) {
        for (int j = i+1; j < n; j++) {

```

```

    for (int k = i+1; k < n; k++) {
        if (j == k) continue;
        double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
        double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
        double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
        bool flag = zn < 0;
        for (int m = 0; flag && m < n; m++)
            flag = flag && ((x[m]-x[i])*xn +
                           (y[m]-y[i])*yn +
                           (z[m]-z[i])*zn <= 0);
        if (flag) ret.push_back(triple(i, j, k));
    }
}

return ret;
}

int main()
{
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //           0 3 2

    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}

```

9 Geometry (ITA)

9.1 Circulo 2d

```

#include <stdio>
#include <cmath>
#include <algorithm> // std::random_shuffle
#include <vector> // std::vector
using namespace std;
#define EPS 1e-9

/*
 * Point 2D
 */

struct point {
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) {}
    point operator +(point other) const{
        return point(x + other.x, y + other.y);
    }
    point operator -(point other) const{
        return point(x - other.x, y - other.y);
    }
    point operator *(double k) const{
        return point(x*k, y*k);
    }
};

double dist(point p1, point p2) {
    return hypot(p1.x - p2.x, p1.y - p2.y);
}

double inner(point p1, point p2) {
    return p1.x*p2.x + p1.y*p2.y;
}

double cross(point p1, point p2) {
    return p1.x*p2.y - p1.y*p2.x;
}

point rotate(point p, double rad) {
    return point(p.x * cos(rad) - p.y * sin(rad),
                p.x * sin(rad) + p.y * cos(rad));
}

point closestToLineSegment(point p, point a, point b) {
    double u = inner(p-a, b-a) / inner(b-a, b-a);
}

```

```

    if (u < 0.0) return a;
    if (u > 1.0) return b;
    return a + ((b-a)*u);
}

double distToLineSegment(point p, point a, point b) {
    return dist(p, closestToLineSegment(p, a, b));
}

/*
 * Circle 2D
 */

struct circle {
    point c;
    double r;
    circle() { c = point(); r = 0; }
    circle(point _c, double _r) : c(_c), r(_r) {}
    double area() { return acos(-1.0)*r*r; }
    double chord(double rad) { return 2*r*sin(rad/2.0); }
    double sector(double rad) { return 0.5*rad*area()/acos(-1.0); }
    bool intersects(circle other) {
        return dist(c, other.c) < r + other.r;
    }
    bool contains(point p) { return dist(c, p) <= r + EPS; }
    pair<point, point> getTangentPoint(point p) {
        double dl = dist(p, c), theta = asin(r/dl);
        point p1 = rotate(c-p, -theta);
        point p2 = rotate(c-p, theta);
        p1 = p1*(sqrt(dl*dl-r*r)/dl)+p;
        p2 = p2*(sqrt(dl*dl-r*r)/dl)+p;
        return make_pair(p1, p2);
    }
};

circle circumcircle(point a, point b, point c) {
    circle ans;
    point u = point((b-a).y, -(b-a).x);
    point v = point((c-a).y, -(c-a).x);
    point n = (c-b)*0.5;
    double t = cross(u,n)/cross(v,u);
    ans.c = ((a+c)*0.5) + (v*t);
    ans.r = dist(ans.c, a);
    return ans;
}

int insideCircle(point p, circle c) {
    if (fabs(dist(p, c.c) - c.r)<EPS) return 1;
    else if (dist(p, c.c) < c.r) return 0;
    else return 2;
} //0 = inside/1 = border/2 = outside

circle incircle(point p1, point p2, point p3) {
    double m1=dist(p2, p3);
    double m2=dist(p1, p3);
    double m3=dist(p1, p2);
    point c = (p1*m1+p2*m2+p3*m3)*(1/(m1+m2+m3));
    double s = 0.5*(m1+m2+m3);
    double r = sqrt(s*(s-m1)*(s-m2)*(s-m3))/s;
    return circle(c, r);
}

//Minimum enclosing circle, O(n)
circle minimumCircle(vector<point> p) {
    random_shuffle(p.begin(), p.end());
    circle C = circle(p[0], 0.0);
    for(int i = 0; i < (int)p.size(); i++) {
        if (C.contains(p[i])) continue;
        C = circle(p[i], 0.0);
        for(int j = 0; j < i; j++) {
            if (C.contains(p[j])) continue;
            C = circle((p[j] + p[i])*0.5, 0.5*dist(p[j], p[i]));
            for(int k = 0; k < j; k++) {
                if (C.contains(p[k])) continue;
                C = circumcircle(p[j], p[i], p[k]);
            }
        }
    }
    return C;
}

/*
 * Codeforces 101707B
 */
/*
point A, B;
circle C;

double getd2(point a, point b) {
}

```

```

double h = dist(a, b);
double r = C.r;
double alpha = asin(h/(2*r));
while (alpha < 0) alpha += 2*acos(-1.0);
return dist(a, A) + dist(b, B) + r*2*min(alpha, 2*acos(-1.0) - alpha);
}

int main() {
    scanf("%lf %lf", &A.x, &A.y);
    scanf("%lf %lf", &B.x, &B.y);
    scanf("%lf %lf %lf", &C.c.x, &C.c.y, &C.r);
    double ans;
    if (distToLineSegment(C.c, A, B) >= C.r) {
        ans = dist(A, B);
    }
    else {
        pair<point, point> tan1 = C.getTangentPoint(A);
        pair<point, point> tan2 = C.getTangentPoint(B);
        ans = 1e+30;
        ans = min(ans, getd2(tan1.first, tan2.first));
        ans = min(ans, getd2(tan1.first, tan2.second));
        ans = min(ans, getd2(tan1.second, tan2.first));
        ans = min(ans, getd2(tan1.second, tan2.second));
    }
    printf("%.18f\n", ans);
    return 0;
}*/

/*
 * Codeforces 101707J
 */

vector<point> P;
int n;

int main () {
    scanf("%d", &n);
    P.resize(n);
    for(int i = 0; i < n; i++) {
        scanf("%lf %lf", &P[i].x, &P[i].y);
    }
    circle ans = minimumCircle(P);
    printf("%.18f %.18f\n%.18f\n", ans.c.x, ans.c.y, ans.r);
    return 0;
}

```

9.2 Minkowski

```

#include <cmath>
#define EPS 1e-9

/*
 * Point 2D
 */

struct point {
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) {}
    bool operator < (point other) const {
        if (fabs(x - other.x) > EPS) return x < other.x;
        else return y < other.y;
    }
    point operator + (point other) const {
        return point(x + other.x, y + other.y);
    }
    point operator - (point other) const {
        return point(x - other.x, y - other.y);
    }
    point operator * (double k) const {
        return point(x*k, y*k);
    }
};

double dist(point p1, point p2) {
    return hypot(p1.x - p2.x, p1.y - p2.y);
}

double inner(point p1, point p2) {
    return p1.x*p2.x + p1.y*p2.y;
}

double cross(point p1, point p2) {
    return p1.x*p2.y - p1.y*p2.x;
}

```

```

bool collinear(point p, point q, point r) {
    return fabs(cross(p-q, r-p)) < EPS;
}

/*
 * Polygon 2D
 */

#include <vector>
#include <algorithm>
using namespace std;

typedef vector<point> polygon;

double signedArea(polygon & P) {
    double result = 0.0;
    int n = P.size();
    for (int i = 0; i < n; i++) {
        result += cross(P[i], P[(i+1)%n]);
    }
    return result / 2.0;
}

int leftmostIndex(vector<point> & P) {
    int ans = 0;
    for(int i=1; i<(int)P.size(); i++) {
        if (P[i] < P[ans]) ans = i;
    }
    return ans;
}

polygon make_polygon(vector<point> P) {
    if (signedArea(P) < 0.0) reverse(P.begin(), P.end());
    int li = leftmostIndex(P);
    reverse(P.begin(), P.begin()+li);
    reverse(P.begin()+li, P.end());
    reverse(P.begin(), P.end());
    return P;
}

/*
 * Minkowski sum
 */

polygon minkowski(polygon & A, polygon & B) {
    polygon P; point v1, v2;
    int n1 = A.size(), n2 = B.size();
    P.push_back(A[0]+B[0]);
    for(int i = 0, j = 0; i < n1 || j < n2; ) {
        v1 = A[(i+1)%n1]-A[i%n1];
        v2 = B[(j+1)%n2]-B[j%n2];
        if (j == n2 || cross(v1, v2) > EPS) {
            P.push_back(P.back() + v1); i++;
        }
        else if (i == n1 || cross(v1, v2) < -EPS) {
            P.push_back(P.back() + v2); j++;
        }
        else {
            P.push_back(P.back() + (v1+v2));
            i++; j++;
        }
    }
    P.pop_back();
    return P;
}

/*
 * Triangle 2D
 */

struct triangle {
    point a, b, c;
    triangle() { a = b = c = point(); }
    triangle(point _a, point _b, point _c) : a(_a), b(_b), c(_c) {}
    int isInside(point p) {
        double u = cross(b-a, p-a)*cross(b-a, c-a);
        double v = cross(c-b, p-b)*cross(c-b, a-b);
        double w = cross(a-c, p-c)*cross(a-c, b-c);
        if (u > 0.0 && v > 0.0 && w > 0.0) return 0;
        if (u < 0.0 || v < 0.0 || w < 0.0) return 2;
        else return 1;
    } //0 = inside/ 1 = border/ 2 = outside
};

int isInsideTriangle(point a, point b, point c, point p) {
    return triangle(a,b,c).isInside(p);
} //0 = inside/ 1 = border/ 2 = outside

```

```
/*
 * Convex query
 */

bool query(polygon &P, point q) {
    int i = 1, j = P.size()-1, m;
    if (cross(P[i]-P[0], P[j]-P[0]) < -EPS)
        swap(i, j);
    while(abs(j-i) > 1) {
        int m = (i+j)/2;
        if (cross(P[m]-P[0], q-P[0]) < 0) j = m;
        else i = m;
    }
    return isInsideTriangle(P[0], P[i], P[j], q) != 2;
}

/*
 * Codeforces 87E
 */
#include <cstdio>

void printpolygon(polygon & P) {
    printf("printing polygon:\n");
    for(int i=0; i<(int)P.size(); i++) {
        printf("%.2f %.2f\n", P[i].x, P[i].y);
    }
}

polygon city[3], P;

int main() {
    double x, y;
    for(int i = 0, n; i < 3; i++) {
        scanf("%d", &n);
        P.clear();
        while(n --> 0) {
            scanf("%lf %lf", &x, &y);
            P.push_back(point(x, y));
        }
        city[i] = make_polygon(P);
    }
    P = minkowski(city[0], city[1]);
    P = minkowski(P, city[2]);

    int m;
    scanf("%d", &m);
    while(m --> 0) {
        scanf("%lf %lf", &x, &y);
        if (query(P, point(x, y)*3.0)) printf("YES\n");
        else printf("NO\n");
    }
    return 0;
}
```

9.3 Great Circle

```
#include <cstdio>
#include <cmath>

double gcTheta(double pLat, double pLong, double qLat, double qLong) {
    pLat += acos(-1.0) / 180.0; pLong += acos(-1.0) / 180.0; // convert degree to radian
    qLat += acos(-1.0) / 180.0; qLong += acos(-1.0) / 180.0;
    return acos(cos(pLat)*cos(pLong)*cos(qLat)+cos(qLong) +
        cos(pLat)*sin(pLong)*cos(qLat)+sin(qLong) +
        sin(pLat)*sin(qLat));
}

double gcDistance(double pLat, double pLong, double qLat, double qLong, double radius) {
    return radius*gcTheta(pLat, pLong, qLat, qLong);
}
```

10 Miscellaneous

10.1 builtin

```
__builtin_ctz(x) // trailing zeroes
__builtin_clz(x) // leading zeroes
__builtin_popcount(x) // # bits set
```

```
__builtin_ffs(x) // index(LSB) + 1 [0 if x==0]

// Add 11 to the end for long long [__builtin_clzll(x)]
```

10.2 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	1009	1013
1019	1021	1031	1033	1039	1049	1051	1061	1063	1069
1087	1091	1093	1097	1103	1109	1117	1123	1129	1151
1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
1229	1231	1237	1249	1259	1277	1279	1283	1289	1291
1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
1381	1399	1409	1423	1427	1429	1433	1439	1447	1451
1453	1459	1471	1481	1483	1487	1489	1493	1499	1511
1523	1531	1543	1549	1553	1559	1567	1571	1579	1583
1597	1601	1607	1609	1613	1619	1621	1627	1637	1657
1663	1667	1669	1693	1697	1699	1709	1721	1723	1733
1741	1747	1753	1759	1777	1783	1787	1789	1801	1811
1823	1831	1847	1861	1867	1871	1873	1877	1879	1889
1901	1907	1913	1931	1933	1949	1951	1973	1979	1987

970'997	971'483	921'281'269	999'279'733
1'000'000'009	1'000'000'021	1'000'000'409	1'005'012'527

10.3 Week day

```
int v[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };
int day(int d, int m, int y) {
    y -= m<3;
    return (y + y/4 - y/100 + y/400 + v[m-1] + d)%7;
}
```

10.4 Date

```
struct Date {
    int d, m, y;
    static int mnt[], mntsum[];

    Date() : d(1), m(1), y(1) {}
    Date(int d, int m, int y) : d(d), m(m), y(y) {}
    Date(int days) : d(1), m(1), y(1) { advance(days); }

    bool bissexto() { return (y%4 == 0 and y%100) or (y%400 == 0); }

    int mdays() { return mnt[m] + (m == 2)*bissexto(); }
    int ydays() { return 365+bissexto(); }

    int msum() { return mntsum[m-1] + (m > 2)*bissexto(); }
    int ysum() { return 365*(y-1) + (y-1)/4 - (y-1)/100 + (y-1)/400; }

    int count() { return (d-1) + msum() + ysum(); }

    int day() {
        int x = y - (m<3);
        return (x + x/4 - x/100 + x/400 + mntsum[m-1] + d + 6)%7;
    }

    void advance(int days) {
        days += count();
        d = m = 1, y = 1 + days/366;
        days -= count();
    }
}
```

```

    while(days >= ydays()) days -= ydays(), y++;
    while(days >= mdays()) days -= mdays(), m++;
    d += days;
}
};

int Date::mnt[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int Date::mntsum[13] = {};
for(int i=1; i<13; ++i) Date::mntsum[i] = Date::mntsum[i-1] + Date::mnt[i];

```

10.5 Latitude Longitude (Stanford)

```

/*
Converts from rectangular coordinates to latitude/longitude and vice
versa. Uses degrees (not radians).
*/

```

```

#include <iostream>
#include <cmath>

using namespace std;

struct ll
{
    double r, lat, lon;
};

struct rect
{
    double x, y, z;
};

ll convert(rect& P)
{
    ll Q;
    Q.r = sqrt(P.x*P.x+P.y*P.y+P.z*P.z);
    Q.lat = 180/M_PI*asin(P.z/Q.r);
    Q.lon = 180/M_PI*acos(P.x/sqrt(P.x*P.x+P.y*P.y));

    return Q;
}

rect convert(ll& Q)
{
    rect P;
    P.x = Q.r*cos(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
    P.y = Q.r*sin(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
    P.z = Q.r*sin(Q.lat*M_PI/180);

    return P;
}

int main()
{
    rect A;
    ll B;

    A.x = -1.0; A.y = 2.0; A.z = -3.0;

    B = convert(A);
    cout << B.r << " " << B.lat << " " << B.lon << endl;

    A = convert(B);
    cout << A.x << " " << A.y << " " << A.z << endl;
}

```

10.6 Python

```

# reopen
import sys
sys.stdout = open('out', 'w')
sys.stdin = open('in', 'r')

```

10.7 Sqrt Decomposition

```

// Square Root Decomposition (Mo's Algorithm) - O(n^(3/2))
const int N = 1e5+1, SQ = 500;
int n, m, v[N];

void add(int p) { /* add value to aggregated data structure */ }
void rem(int p) { /* remove value from aggregated data structure */ }

struct query { int i, l, r, ans; } qs[N];

bool c1(query a, query b) {
    if(a.l/SQ != b.l/SQ) return a.l < b.l;
    return a.l/SQ&1 ? a.r > b.r : a.r < b.r;
}

bool c2(query a, query b) { return a.i < b.i; }

/* inside main */
int l = 0, r = -1;
sort(qs, qs+m, c1);
for (int i = 0; i < m; ++i) {
    query &q = qs[i];
    while (r < q.r) add(v[++r]);
    while (r > q.r) rem(v[r--]);
    while (l < q.l) rem(v[l++]);
    while (l > q.l) add(v[--l]);

    q.ans = /* calculate answer */;
}

sort(qs, qs+m, c2); // sort to original order

```

10.8 Bitset

```

//Goes through the subsets of a set x :
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);

```

10.9 Parenthesis to Polish (ITA)

```

#include <cstdio>
#include <map>
#include <stack>
using namespace std;

/*
* Parenthetic to polish expression conversion
*/

inline bool isOp(char c) {
    return c=='+' || c=='-' || c=='*' || c=='/' || c=='^';
}

inline bool isCarac(char c) {
    return (c>='a' && c<='z') || (c>='A' && c<='Z') || (c>='0' && c<='9');
}

int paren2polish(char* paren, char* polish) {
    map<char, int> prec;
    prec['('] = 0;
    prec['+'] = prec['-'] = 1;
    prec['*'] = prec['/'] = 2;
    prec['^'] = 3;
    int len = 0;
    stack<char> op;
    for (int i = 0; paren[i]; i++) {
        if (isOp(paren[i])) {
            while (!op.empty() && prec[op.top()] >= prec[paren[i]]) {
                polish[len++] = op.top(); op.pop();
            }
            op.push(paren[i]);
        }
        else if (paren[i]=='(') op.push('(');
        else if (paren[i]==')') {
            for (; op.top()!='('; op.pop())
                polish[len++] = op.top();
            op.pop();
        }
        else if (isCarac(paren[i]))

```

```

        polish[len++] = paren[i];
    }
    for(; !op.empty(); op.pop())
        polish[len++] = op.top();
    polish[len] = 0;
    return len;
}

/*
 * TEST MATRIX
 */

int main() {
    int N, len;
    char polish[400], paren[400];
    scanf("%d", &N);
    for (int j=0; j<N; j++) {
        scanf("%s", paren);
        paren2polish(paren, polish);
        printf("%s\n", polish);
    }
    return 0;
}

```

11 Math Extra

11.1 Combinatorial formulas

$$\begin{aligned}
 \sum_{k=0}^n k^2 &= n(n+1)(2n+1)/6 \\
 \sum_{k=0}^n k^3 &= n^2(n+1)^2/4 \\
 \sum_{k=0}^n k^4 &= (6n^5 + 15n^4 + 10n^3 - n)/30 \\
 \sum_{k=0}^n k^5 &= (2n^6 + 6n^5 + 5n^4 - n^2)/12 \\
 \sum_{k=0}^n x^k &= (x^{n+1} - 1)/(x - 1) \\
 \sum_{k=0}^n kx^k &= (x - (n+1)x^{n+1} + nx^{n+2})/(x - 1)^2 \\
 \binom{n}{k} &= \frac{n!}{(n-k)!k!} \\
 \binom{n}{k} &= \binom{n-1}{k} + \binom{n-1}{k-1} \\
 \binom{n}{k} &= \frac{n}{n-k} \binom{n-1}{k} \\
 \binom{n}{k} &= \frac{n-k+1}{k} \binom{n}{k-1} \\
 \binom{n+1}{k} &= \frac{n+1}{n-k+1} \binom{n}{k} \\
 \binom{n}{k+1} &= \frac{n-k}{k+1} \binom{n}{k} \\
 \sum_{k=1}^n k \binom{n}{k} &= n2^{n-1} \\
 \sum_{k=1}^n k^2 \binom{n}{k} &= (n + n^2)2^{n-2} \\
 \binom{m+n}{r} &= \sum_{k=0}^r \binom{m}{k} \binom{n}{r-k} \\
 \binom{n}{k} &= \prod_{i=1}^k \frac{n-k+i}{i}
 \end{aligned}$$

11.2 Number theory identities

Lucas' Theorem: For non-negative integers m and n and a prime p ,

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{n_i} \pmod{p},$$

where

$$m = m_k p^k + m_{k-1} p^{k-1} + \cdots + m_1 p + m_0$$

is the base p representation of m , and similarly for n .

11.3 Stirling Numbers of the second kind

Number of ways to partition a set of n numbers into k non-empty subsets.

$$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = \frac{1}{k!} \sum_{j=0}^k (-1)^{(k-j)} \binom{k}{j} j^n$$

Recurrence relation:

$$\left\{ \begin{matrix} 0 \\ 0 \end{matrix} \right\} = 1$$

$$\left\{ \begin{matrix} n \\ 0 \end{matrix} \right\} = \left\{ \begin{matrix} 0 \\ n \end{matrix} \right\} = 1$$

$$\left\{ \begin{matrix} n+1 \\ k \end{matrix} \right\} = k \left\{ \begin{matrix} n \\ k \end{matrix} \right\} + \left\{ \begin{matrix} n \\ k-1 \end{matrix} \right\}$$

11.4 Burnside's Lemma

Let G be a finite group that acts on a set X . For each g in G let X^g denote the set of elements in X that are fixed by g , which means $X^g = \{x \in X | g(x) = x\}$. Burnside's lemma asserts the following formula for the number of orbits, denoted $|X/G|$:

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

11.5 Numerical integration

RK4: to integrate $\dot{y} = f(t, y)$ with $y_0 = y(t_0)$, compute

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1)$$

$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2)$$

$$k_4 = f(t_n + h, y_n + hk_3)$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

	S	R	X	Assunto	Descricao	Diff
A						
B						
C						
D						
E						
F						
G						
H						
I						
J						
K						
L						