ClosestPair

DivideAndConquerOnTree

SegmentTreeAndHeavyLightDecomposition

Dijkstra

Hungarian

StronglyConnectedComponent\_Kosaraju

MaxFlow\_Dinic

MinCostFlow\_Dinic

Splay

KMP

ExtendKMP

SuffixArray

TrieGraph

LongestPalindromicSubstring

GaussianElimination

FFT

NTT

PrimeAndPhiAndMu

MöbiusInversionFormula

PolicyBasedDataStructure\_RBTree

# ClosestPair

#include <cstdio>

#include <cmath>

#include <algorithm>

using namespace std;

const int MAXN = 100005, MAXLOGN = 17;

typedef double coord\_t;

int N;

struct Point {

coord\_t x, y;

int xRank;

}p[MAXN];

int id[MAXLOGN][MAXN];

bool cmpX(const Point &A, const Point &B) {

return A.x < B.x;

}

bool cmpY(int i, int j) {

return p[i].y < p[j].y;

}

double dist(const Point &A, const Point &B) {

return sqrt((A.x - B.x) \* (A.x - B.x) + (A.y - B.y) \* (A.y - B.y));

}

double closestPairDist(int l, int r, int depth) {

if (l + 2 == r) return dist(p[l], p[l + 1]);

if (l + 3 == r) return min(dist(p[l], p[l + 1]), min(dist(p[l], p[l + 2]), dist(p[l + 1], p[l + 2])));

int m = (l + r) >> 1, j = l, k = m;

for (int i = l; i < r; ++i)

if (id[depth][i] < m) id[depth + 1][j++]= id[depth][i];

else id[depth + 1][k++] = id[depth][i];

double d = min(closestPairDist(l, m, depth + 1), closestPairDist(m, r, depth + 1));

j = l;

for (int i = l; i < r; ++i)

if (p[id[depth][i]].x > p[m].x - d && p[id[depth][i]].x < p[m].x + d)

id[depth][j++] = id[depth][i];

for (int i = l; i < j; ++i) {

for (k = i + 1; k <= i + 7 && k < j; ++k)

d = min(d, dist(p[id[depth][i]], p[id[depth][k]]));

}

return d;

}

void solve() {

sort(p, p + N, cmpX);

for (int i = 0; i < N; ++i) id[0][i] = i;

sort(id[0], id[0] + N, cmpY);

printf("%.2f\n", closestPairDist(0, N, 0));

}

int main() {

scanf("%d", &N);

while (N > 0) {

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

scanf("%lf%lf", &p[i].x, &p[i].y);

solve();

scanf("%d", &N);

}

return 0;

}

# DivideAndConquerOnTree

struct Edge {

Edge(int end, int len): end(end), len(len) {}

int end, len;

};

struct Vertex {

bool visited;

vector<Edge> eList;

int size;

}v[MAXN];

struct CenterStruct{

void dfs(int x, int f) {

v[x].size = 1;

int balance = 0;

int s = v[x].eList.size();

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

int y = v[x].eList[i].end;

if (!v[y].visited && y != f) {

dfs(y, x);

v[x].size += v[y].size;

balance = max(balance, v[y].size);

}

}

balance = max(balance, size - v[x].size);

if (balance < minBalance)

minBalance = balance, center = x;

}

int getCenter(int s, int x) {

minBalance = size = s;

dfs(x, -1);

return center;

}

int size, minBalance, center;

}centerStruct;

solve(centerStruct.getCenter(N, 1));

# SegmentTreeAndHeavyLightDecomposition

// SPOJ - Can you answer these queries VII

#include <cstdio>

#include <vector>

#include <algorithm>

using namespace std;

const int MAXN = 100005, MAXL = 131072, MAXLOGN = 25; // L is the lowest power of 2 greater than or equal to N

const int NO\_MARK = 0x3fffffff;

struct Tree {

int N;

struct Node {

int id, depth, size, val;

Node \*father, \*hSon, \*top;

vector<Node \*> neighbors;

}nodes[MAXN];

void addEdge(int x, int y) {

nodes[x].neighbors.push\_back(&nodes[y]);

nodes[y].neighbors.push\_back(&nodes[x]);

}

}tree;

struct SegmentTree {

struct Info {

Info(): sum(0), maxSum(0), lMaxSum(0), rMaxSum(0), mark(NO\_MARK) {}

int sum, maxSum, lMaxSum, rMaxSum, mark;

void merge(Info lSon, Info rSon) {

// To be filled in

sum = lSon.sum + rSon.sum;

lMaxSum = max(lSon.lMaxSum, lSon.sum + rSon.lMaxSum);

rMaxSum = max(rSon.rMaxSum, rSon.sum + lSon.rMaxSum);

maxSum = max(max(lSon.maxSum, rSon.maxSum), lSon.rMaxSum + rSon.lMaxSum);

}

Info reverse() {

// To be filled in

Info ans = \*this;

swap(ans.lMaxSum, ans.rMaxSum);

return ans;

}

}nodes[MAXL << 1];

int L;

void buildTree() {

L = 1;

while (L <= tree.N) L <<= 1;

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

// To be filled in

nodes[L + tree.nodes[i].id].sum = tree.nodes[i].val;

nodes[L + tree.nodes[i].id].maxSum

= nodes[L + tree.nodes[i].id].lMaxSum

= nodes[L + tree.nodes[i].id].rMaxSum = max(tree.nodes[i].val, 0);

nodes[L + tree.nodes[i].id].mark = NO\_MARK;

}

for (int i = tree.N; i < L; ++i)

nodes[L + i] = Info();

for (int i = L - 1; i >= 1; --i)

nodes[i].mark = NO\_MARK, nodes[i].merge(nodes[i << 1], nodes[(i << 1) | 1]);

}

void paint(int id, int l, int r, int val) { // Paint a node (for range modification only)

// To be filled in

nodes[id].sum = val \* (r - l);

nodes[id].maxSum = nodes[id].lMaxSum = nodes[id].rMaxSum = max(0, nodes[id].sum);

nodes[id].mark = val;

}

void pushDown(int id, int l, int r) {

if (id >= L) return;

if (NO\_MARK == nodes[id].mark) return;

paint(id << 1, l, (l + r) >> 1, nodes[id].mark);

paint((id << 1) | 1, (l + r) >> 1, r, nodes[id].mark);

nodes[id].mark = NO\_MARK;

}

void modify(int id, int l, int r, int p, int q, int val) {

if (l == p && r == q) {

paint(id, l, r, val);

return;

}

int m = (l + r) >> 1;

pushDown(id, l, r);

if (q <= m) modify(id << 1, l, m, p, q, val);

else if (p >= m) modify((id << 1) | 1, m, r, p, q, val);

else modify(id << 1, l, m, p, m, val), modify((id << 1) | 1, m, r, m, q, val);

nodes[id].merge(nodes[id << 1], nodes[(id << 1) | 1]);

}

/\*void modify(int x, int val) { // Single-node modification

// To be filled in

nodes[L + x].val += val;

for (int i = (L + x) >> 1; i >= 1; i >>= 1)

nodes[i].merge(nodes[i << 1], nodes[(i << 1) | 1]);

}\*/

Info query(int id, int l, int r, int p, int q) { // nodes[id] represents the interval [l, r); query the interval [p, q)

if (p == l && r == q) {

return nodes[id];

}

pushDown(id, l, r);

int m = (l + r) >> 1;

if (q <= m) return query(id << 1, l, m, p, q);

if (p >= m) return query((id << 1) | 1, m, r, p, q);

Info ans;

ans.merge(query(id << 1, l, m, p, m), query((id << 1) | 1, m, r, m, q));

return ans;

}

}segTree;

struct HeavyLightDecomposition {

int cnt;

void dfs(Tree::Node \*x) { // First-round DFS; in particular, find the heavy sons

x->size = 1;

int s = x->neighbors.size();

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

Tree::Node \*y = x->neighbors[i];

if (y != x->father) {

y->father = x;

y->depth = x->depth + 1;

dfs(y);

if (NULL == x->hSon || x->hSon->size < y->size)

x->hSon = y;

x->size += y->size;

}

}

}

void dfs2(Tree::Node \*x, Tree::Node \*t) { // Put the tree nodes into the segment tree

x->id = cnt++;

x->top = t;

if (NULL != x->hSon) dfs2(x->hSon, t);

int s = x->neighbors.size();

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

Tree::Node \*y = x->neighbors[i];

if (y != x->father && y != x->hSon) dfs2(y, y);

}

}

void modify(int x, int y, int val) {

Tree::Node \*u = &tree.nodes[x], \*v = &tree.nodes[y];

Tree::Node \*tu = u->top, \*tv = v->top;

while (tu != tv) {

// Be careful whether reversions are required here

if (tu->depth > tv->depth) {

segTree.modify(1, 0, segTree.L, tu->id, u->id + 1, val);

u = tu->father;

tu = u->top;

}

else {

segTree.modify(1, 0, segTree.L, tv->id, v->id + 1, val);

v = tv->father;

tv = v->top;

}

}

if (u->depth <= v->depth) segTree.modify(1, 0, segTree.L, u->id, v->id + 1, val);

else segTree.modify(1, 0, segTree.L, v->id, u->id + 1, val);

}

int query(int x, int y) {

SegmentTree::Info infos1[MAXLOGN], infos2[MAXLOGN], ans;

int cnt1 = 0, cnt2 = 0;

Tree::Node \*u = &tree.nodes[x], \*v = &tree.nodes[y];

Tree::Node \*tu = u->top, \*tv = v->top;

while (tu != tv) {

if (tu->depth > tv->depth) {

infos1[cnt1++] = segTree.query(1, 0, segTree.L, tu->id, u->id + 1).reverse();

u = tu->father;

tu = u->top;

}

else {

infos2[cnt2++] = segTree.query(1, 0, segTree.L, tv->id, v->id + 1);

v = tv->father;

tv = v->top;

}

}

if (u->depth <= v->depth)

infos1[cnt1++] = segTree.query(1, 0, segTree.L, u->id, v->id + 1);

else

infos1[cnt1++] = segTree.query(1, 0, segTree.L, v->id, u->id + 1).reverse();

for (int i = 0; i < cnt1; ++i) ans.merge(ans, infos1[i]);

for (int i = cnt2 - 1; i >= 0; --i) ans.merge(ans, infos2[i]);

return ans.maxSum;

}

void decompose() {

tree.nodes[1].depth = 0;

dfs(&tree.nodes[1]);

dfs2(&tree.nodes[1], &tree.nodes[1]);

segTree.buildTree();

}

}hld;

void init() { // Initialize tree and hld. segTree is initialized in SegmentTree::buildTree().

for (int i = 1; i <= tree.N; ++i)

tree.nodes[i].hSon = NULL, tree.nodes[i].neighbors.clear();

hld.cnt = 0;

}

int main() {

int Q, op, x, y, val;

scanf("%d", &tree.N);

init();

for (int i = 1; i <= tree.N; ++i)

scanf("%d", &tree.nodes[i].val);

for (int i = 1; i < tree.N; ++i)

scanf("%d%d", &x, &y), tree.addEdge(x, y);

hld.decompose();

scanf("%d", &Q);

while (Q--) {

scanf("%d", &op);

if (1 == op) scanf("%d%d", &x, &y), printf("%d\n", hld.query(x, y));

else scanf("%d%d%d", &x, &y, &val), hld.modify(x, y, val);

}

return 0;

}

# Dijkstra

#include <queue>

#include <vector>

#include <utility>

using namespace std;

const int MAXN = 100005, INFTY = 0x3fffffff;

int N, d[MAXN];

struct Edge {

Edge(int end = 0, int cost = 0): end(end), cost(cost) {}

int end, cost;

};

vector<Edge> eList[MAXN];

bool ok[MAXN];

void addEdge(int begin, int end, int cost) {

eList[begin].push\_back(Edge(end, cost));

}

void Dijkstra() {

int S = 1; // Source

priority\_queue<pair<int, int> > Q;

Q.push(make\_pair(0, S));

for (int i = 1; i <= N; ++i) d[i] = INFTY, ok[i] = false;

d[S] = 0;

while (!Q.empty()) {

pair<int, int> p = Q.top();

Q.pop();

int x = p.second;

if (ok[x]) continue;

ok[x] = true;

int s = eList[x].size();

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

int y = eList[x][i].end, c = eList[x][i].cost;

if (d[x] + c < d[y])

d[y] = d[x] + c, Q.push(make\_pair(-d[y], y));

}

}

}

# Hungarian

const int MAXN = 205;

// Number of vertices on each side.

int NX, NY;

// Companions. cx[i] (cy[i]) is the companion of the i-th vertex on the left (right).

int cx[MAXN], cy[MAXN];

// G stores the graph. G[i][j] == 1 iff the i-th vertex on the left and the j-th vertex on the right are connected.

bool G[MAXN][MAXN], visited[MAXN];

int dfs(int u) {

for (int v = 1; v <= NY; ++v) {

if (G[u][v] && !visited[v]) {

visited[v] = 1;

if (cy[v] == -1 || dfs(cy[v])) {

cx[u] = v;

cy[v] = u;

return 1;

}

}

}

return 0;

}

int maxMatch() {

int ans = 0;

memset(cx, -1, sizeof(cx));

memset(cy, -1, sizeof(cy));

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

if (cx[i] == -1)

memset(visited, 0, sizeof(visited)), ans += dfs(i);

return ans;

}

# StronglyConnectedComponent\_Kosaraju

#include <cstring>

#include <vector>

#include <algorithm>

using namespace std;

const int MAXN = 100005;

int N; // Number of Vertices

vector<int> eList[MAXN], reList[MAXN]; // Edge list and reverse edge list

vector<int> vList;

bool visited[MAXN];

int sccId[MAXN];

void addEdge(int x, int y) {

eList[x].push\_back(y);

reList[y].push\_back(x);

}

void DFS(int x) {

visited[x] = true;

int s = eList[x].size();

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

if (!visited[eList[x][i]])

DFS(eList[x][i]);

vList.push\_back(x);

}

void RDFS(int x, int id) {

visited[x] = true;

sccId[x] = id;

int s = reList[x].size();

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

if (!visited[reList[x][i]])

RDFS(reList[x][i], id);

}

int findSCC() { // Returns the number of strongly connected components

memset(visited, 0, sizeof(visited));

vList.clear();

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

if (!visited[i])

DFS(i);

memset(visited, 0, sizeof(visited));

int nScc = 0;

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

if (!visited[vList[i]])

RDFS(vList[i], ++nScc);

return nScc;

}

# MaxFlow\_Dinic

#include <cstdio>

#include <queue>

using namespace std;

typedef int flow\_t;

const int MAXN = 505, MAXM = 100005, DIRECTED = 0, UNDIRECTED = 1;

const flow\_t INFTY = 0x3fffffff;

int N, S, T, now;

struct edge {

flow\_t remain;

int endVertexId, nextEdgeId;

}e[MAXM << 1];

struct vertex {

int firstEdgeId, level, firstUnsaturEdgeId;

}v[MAXN];

void \_addEdge(int begin, int end, flow\_t c) {

e[now].remain = c;

e[now].endVertexId = end;

e[now].nextEdgeId = v[begin].firstEdgeId;

v[begin].firstEdgeId = now++;

}

void addEdge(int begin, int end, flow\_t c, int edgeType) {

\_addEdge(begin, end, c);

\_addEdge(end, begin, edgeType \* c);

}

void init() {

now = 0;

for (int i = 0; i < N; ++i) v[i].firstEdgeId = -1;

}

bool markLevel(){

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

v[i].level = -1, v[i].firstUnsaturEdgeId = v[i].firstEdgeId;

v[S].level = 0;

queue<int> Q;

Q.push(S);

while (!Q.empty()) {

int x = Q.front();

Q.pop();

for (int i = v[x].firstEdgeId; i >= 0; i = e[i].nextEdgeId)

if (e[i].remain && v[e[i].endVertexId].level < 0)

v[e[i].endVertexId].level = v[x].level + 1, Q.push(e[i].endVertexId);

}

return v[T].level > 0;

}

flow\_t extendFlow(int x, flow\_t flow) {

if (x == T) return flow;

flow\_t t, total = 0;

for (int &i = v[x].firstUnsaturEdgeId; i >= 0; i = e[i].nextEdgeId) { // Reference!

if (v[e[i].endVertexId].level == v[x].level + 1 && e[i].remain) {

if (t = extendFlow(e[i].endVertexId, min(flow, e[i].remain)))

e[i].remain -= t, e[i ^ 1].remain += t, flow -= t, total += t;

if (0 == flow) break;

}

}

return total;

}

flow\_t Dinic() {

flow\_t flow, total = 0;

while (markLevel())

while (flow = extendFlow(S, INFTY))

total += flow;

return total;

}

void buildGraph() {

// Assign N (number of vertices), S (source) and T (sink) here.

// Vertices are numbered from 0 to N - 1. Hence S and T should be in [0, N).

init();

// Add edges here

}

int main() {

int nCase, n, m;

scanf("%d", &nCase);

while (nCase--) {

scanf("%d%d", &n, &m);

buildGraph();

flow\_t ans = Dinic();

}

return 0;

}

# MinCostFlow\_Dinic

#include <cstdio>

#include <queue>

using namespace std;

typedef int flow\_t, cost\_t;

const int MAXN = 405, MAXM = 1505, DIRECTED = 0, UNDIRECTED = 1;

const flow\_t FLOW\_INFTY = 0x3fffffff;

const cost\_t COST\_INFTY = 0x3fffffff;

int N, S, T, now, K;

bool inQ[MAXN];

struct edge {

flow\_t remain;

cost\_t cost;

int endVertexId, nextEdgeId;

}e[MAXM << 1];

struct vertex {

int firstEdgeId, firstUnsaturEdgeId;

cost\_t level;

}v[MAXN];

void \_addEdge(int begin, int end, flow\_t c, cost\_t w) {

e[now].remain = c;

e[now].cost = w;

e[now].endVertexId = end;

e[now].nextEdgeId = v[begin].firstEdgeId;

v[begin].firstEdgeId = now++;

}

void addEdge(int begin, int end, flow\_t c, int edgeType, cost\_t w = 1) {

\_addEdge(begin, end, c, w);

\_addEdge(end, begin, edgeType \* c, -w);

}

void init() {

now = 0;

for (int i = 0; i < N; ++i) v[i].firstEdgeId = -1, inQ[i] = false;

}

bool markLevel(){ // SPFA

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

v[i].level = COST\_INFTY, v[i].firstUnsaturEdgeId = v[i].firstEdgeId, inQ[i] = false;

v[S].level = 0;

queue<int> Q;

Q.push(S);

inQ[S] = true;

while (!Q.empty()) {

int x = Q.front();

Q.pop();

inQ[x] = false;

for (int i = v[x].firstEdgeId; i >= 0; i = e[i].nextEdgeId) {

if (e[i].remain && v[e[i].endVertexId].level > v[x].level + e[i].cost) {

v[e[i].endVertexId].level = v[x].level + e[i].cost;

if (!inQ[e[i].endVertexId])

Q.push(e[i].endVertexId), inQ[e[i].endVertexId] = true;

}

}

}

return v[T].level < COST\_INFTY;

}

flow\_t extendFlow(int x, flow\_t flow) {

if (x == T) return flow;

inQ[x] = true;

flow\_t t, total = 0;

for (int &i = v[x].firstUnsaturEdgeId; i >= 0; i = e[i].nextEdgeId) { // Reference!

if (v[e[i].endVertexId].level == v[x].level + e[i].cost && e[i].remain && !inQ[e[i].endVertexId]) {

if (t = extendFlow(e[i].endVertexId, min(flow, e[i].remain)))

e[i].remain -= t, e[i ^ 1].remain += t, flow -= t, total += t;

if (0 == flow) break;

}

}

inQ[x] = false;

return total;

}

flow\_t Dinic() {

flow\_t flow, total = 0;

cost\_t cost = 0;

while (markLevel())

while (flow = extendFlow(S, FLOW\_INFTY))

total += flow, cost += flow \* v[T].level;

return cost; // Return total in max flow; return cost in min cost max flow

}

void buildGraph() {

// Assign N (number of vertices), S (source) and T (sink) here.

// Vertices are numbered from 0 to N - 1. Hence S and T should be in [0, N).

init();

// Add edges here

}

# Splay

#include <cstdio>

const int MAXN = 200005;

struct Splay {

Splay \*child[2], \*father;

int key, size, added;

bool reversed;

}\*root, T[MAXN];

void refresh(Splay \*x) {

x->size = 1;

if (x->child[0] != NULL) x->size += x->child[0]->size;

if (x->child[1] != NULL) x->size += x->child[1]->size;

// Refresh other information here

}

void pushDown(Splay \*x) {

// Push down the labels on x

if (x->reversed) {

Splay \*t = x->child[0];

x->child[0] = x->child[1];

x->child[1] = t;

if (x->child[0] != NULL) x->child[0]->reversed ^= 1;

if (x->child[1] != NULL) x->child[1]->reversed ^= 1;

x->reversed = 0;

}

if (x->added != 0) {

x->key += x->added;

if (x->child[0] != NULL) x->child[0]->added += x->added;

if (x->child[1] != NULL) x->child[1]->added += x->added;

x->added = 0;

}

}

void rotate(Splay \*x, bool dir) {

// x != NULL, and x->father != NULL

/\* y x

/ \ rotate(x, 0) / \

o x -------------> y o

/ \ <------------- / \

o o rotate(y, 1) o o \*/

Splay \*y = x->father;

pushDown(y);

pushDown(x);

y->child[!dir]=x->child[dir];

if (x->child[dir] != NULL) x->child[dir]->father = y;

x->father = y->father;

if (y->father != NULL)

if (y->father->child[0] == y) y->father->child[0] = x;

else y->father->child[1] = x;

x->child[dir] = y;

y->father = x;

if (y == root) root = x;

refresh(y);

refresh(x);

}

void splay(Splay \*x, Splay \*f) {

if (x != NULL) pushDown(x);

if (x == f || x == NULL) return;

while (x->father != f) {

if (x->father->father == f) {

pushDown(x->father);

pushDown(x);

rotate(x, x->father->child[0] == x);

}

else {

Splay \*y = x->father;

Splay \*z = y->father;

pushDown(z);

pushDown(y);

pushDown(x);

if (z->child[0] == y)

if (y->child[0] == x)

rotate(y, 1), rotate(x, 1);

else

rotate(x, 0), rotate(x, 1);

else

if (y->child[0] == x)

rotate(x, 1), rotate(x, 0);

else

rotate(y, 0), rotate(x, 0);

}

}

if (f == NULL) root = x;

// if (f != NULL) refresh(f); // Is it useful?

}

void insertAfter(Splay \*x, Splay \*y) { // Insert x after y

// You should guarantee y != NULL

splay(y, NULL); // Used to push down the labels along the path from the root to y!

Splay \*z = y->child[1];

if (z == NULL) {

y->child[1] = x;

x->father = y;

refresh(y);

}

else {

pushDown(z);

while (z->child[0] != NULL)

z = z->child[0], pushDown(z);

z->child[0] = x;

x->father = z;

while (z != NULL)

refresh(z), z = z->father;

}

splay(x, NULL);

}

Splay \*selectKth(int k) { // Return the k-th element (indexing from 0)

Splay \*tree = root, \*last;

while (tree != NULL) {

pushDown(tree);

int leftSize = (tree->child[0] != NULL) ? tree->child[0]->size : 0;

last = tree;

if (leftSize == k) {

splay(tree, NULL);

return tree;

}

else if (leftSize > k) tree = tree->child[0];

else k -= leftSize + 1, tree = tree->child[1];

}

splay(last, NULL);

return NULL; // K-th element does not exist (the tree has no greater than k elements)

}

Splay \*neighbor(Splay \*x, bool dir) {

splay(x, NULL); // Used to push down the labels along the path from the root to x!

if (x->child[dir] == NULL) return NULL;

x = x->child[dir];

pushDown(x);

while (x->child[!dir] != NULL) x = x->child[!dir], pushDown(x);

return x;

}

Splay \*prev(Splay \*x) {

return neighbor(x, 0);

}

Splay \*succ(Splay \*x) {

return neighbor(x, 1);

}

void del(Splay \*x) { // Delete x from the tree

splay(x, NULL);

if (x->child[0] == NULL) {

root = x->child[1];

if (x->child[1] != NULL) x->child[1]->father = NULL;

}

else {

Splay \*y = prev(x);

splay(y, x);

y->child[1] = x->child[1];

y->father = NULL;

if (x->child[1] != NULL) x->child[1]->father = y;

root = y;

refresh(y);

}

}

int rank(Splay \*x) { // Return the ranking of x (indexing from 0)

splay(x, NULL);

if (x->child[0] == NULL) return 0;

return x->child[0]->size;

}

void add(int l, int r, int val) { // Add val to every element in [l, r)

if (l > 0 && r < N) {

Splay \*x = selectKth(l - 1), \*y = selectKth(r);

splay(x, NULL);

splay(y, x);

if (y->child[0] != NULL)

y->child[0]->added += val;

}

else if (l == 0 && r == N) {

root->added += val;

}

else if (l == 0) {

Splay \*x = selectKth(r);

splay(x, NULL);

if (x->child[0] != NULL)

x->child[0]->added += val;

}

else {

Splay \*x = selectKth(l - 1);

splay(x, NULL);

if (x->child[1] != NULL)

x->child[1]->added += val;

}

}

# KMP

#include <cstdio>

#include <cstring>

const int LEN\_PATTERN = 10005, LEN\_TEXT = 1000005;

int next[LEN\_PATTERN], matchTo[LEN\_TEXT];

char s[LEN\_PATTERN], t[LEN\_TEXT]; // s: pattern string; t: text

// Note that both strings should be indexed from 1

void KMP() {

int lenS = strlen(s + 1), lenT = strlen(t + 1), p = 0;

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

while (p && s[i] != s[p + 1]) p = next[p];

next[i] = (s[i] == s[p + 1]) ? ++p : 0;

}

p = 0;

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

while (p && t[i] != s[p + 1]) p = next[p];

matchTo[i] = (t[i] == s[p + 1]) ? ++p : 0;

}

}

# ExtendedKMP

#include <cstdio>

#include <cstring>

const int LEN\_TEXT = 200005, LEN\_PATTERN = 100005;

char s[LEN\_TEXT], t[LEN\_PATTERN];

int extend[LEN\_TEXT], next[LEN\_PATTERN];

void CalcNext() {

int i = 0, j, po, len = strlen(t);

// Calculate next[0]

next[0] = len;

// Calculate next[1]

while (i + 1 < len && t[i] == t[i + 1]) ++i;

next[1] = i;

po = 1;

// Calculate next[2] ~ next[len - 1]

for (i = 2; i < len; ++i) {

if (i + next[i - po] < po + next[po])

next[i] = next[i - po];

else {

j = (po + next[po]) - i;

if (j < 0) j = 0;

while (i + j < len && t[j] == t[i + j]) ++j;

next[i] = j;

po = i;

}

}

}

void ExtendKMP() {

int i = 0, j, po, lenS = strlen(s), lenT = strlen(t);

while (i < lenS && i < lenT && s[i] == t[i]) ++i;

extend[0] = i;

po = 0;

for (i = 1; i < lenS; ++i) {

if (i + next[i - po] < po + extend[po])

extend[i] = next[i - po];

else {

j = (po + extend[po]) - i;

if (j < 0) j = 0;

while (i + j < lenS && t[j] == s[i + j]) ++j;

extend[i] = j;

po = i;

}

}

}

# SuffixArray

#include <cstdio>

#include <cstring>

const int MAXN = 100005;

typedef int ch\_t; // Type of characters in the string

ch\_t s[MAXN];

int N, r[MAXN], SA[MAXN], height[MAXN], h[MAXN];

struct SuffixArray {

int cnt[MAXN + 128], tmp[MAXN], newR[MAXN]; // Ensure that the size of cnt is not only greater than the maximum possible character (used in the first round) but also at least MAXN (used in the following loop)

void work(const ch\_t s[], int SA[], int r[], int height[], int h[]) {

// s: the input string

// SA, r, height, h: output

// SA[i] is the index of the i-th smallest suffix

// r[i] is the ranking of suffix substring s[i .. N - 1]

// height[i] is the length of longest common prefix of i-th and (i-1)-th smallest suffix

// h[i] = height[r[i]]

int limit = 128; // Values of characters in the string is in [0, limit)

N = strlen(s); // If ch\_t is not char, define another way to find the length!

// Calculate SA and r

memset(cnt, 0, sizeof(int) \* limit);

for (int i = 0; i < N; ++i) ++cnt[r[i] = s[i]];

for (int i = 1; i < limit; ++i) cnt[i] += cnt[i - 1];

for (int i = N - 1; i >= 0; --i) SA[--cnt[s[i]]] = i;

for (int k = 1; k <= N; k <<= 1) {

int j = 0;

for (int i = N - 1; i >= N - k; --i) tmp[j++] = i;

for (int i = 0; i < N; ++i) if (SA[i] >= k) tmp[j++] = SA[i] - k;

memset(cnt, 0, sizeof(int) \* limit);

for (int i = 0; i < N; ++i) ++cnt[r[i]];

for (int i = 1; i < limit; ++i) cnt[i] += cnt[i - 1];

for (int i = N - 1; i >= 0; --i) SA[--cnt[r[tmp[i]]]] = tmp[i];

newR[SA[0]] = 0;

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

newR[SA[i]] = (SA[i] + k < N && SA[i - 1] + k < N

&& r[SA[i]] == r[SA[i - 1]] && r[SA[i] + k] == r[SA[i - 1] + k])

? newR[SA[i - 1]] : newR[SA[i - 1]] + 1;

for (int i = 0; i < N; ++i) r[i] = newR[i];

limit = r[SA[N - 1]] + 1;

if (limit >= N) break;

}

// Calculate h

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

if (0 == r[i])

h[i] = 0;

else

for (h[i] = ((0 == i || 0 == h[i - 1]) ? 0 : (h[i - 1] - 1));

s[i + h[i]] == s[SA[r[i] - 1] + h[i]]; ++h[i]);

}

// Calculate height

for (int i = 0; i < N; ++i) height[i] = h[SA[i]];

}

}suffixArray;

// Usage: suffixArray.work(s, SA, r, height, h);

// Concatenation: when inputting more than 1 string, concatenate them and update height and h like this

const int MAXNSTR = 105;

const ch\_t DELIMITER = '{'; // A character greater than 'a' to 'z' and 'A' to 'Z'

int nStr, lenSuffix[MAXN], id[MAXN];

int main() {

int t = 0;

scanf("%d", &nStr);

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

scanf(" %s", &s[t]);

int len = strlen(&s[t]);

while (s[t]) {

lenSuffix[t] = len--;

id[t++] = i;

}

lenSuffix[t] = 0;

id[t] = -1;

s[t++] = DELIMITER;

}

s[t] = '\0';

suffixArray.work(s, SA, r, height, h);

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

height[i] = min(height[i], lenSuffix[SA[i]]);

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

h[i] = height[r[i]];

return 0;

}

# TrieGraph

#include <cstdio>

#include <cstring>

#include <queue>

using namespace std;

const int MAX\_NODES = 1000005, LEN\_PATTERN = 1000005, LEN\_TEXT = 1000005, NUM\_LETTER = 26;

inline int char2int(char c) { // Map a valid character to an integer in [0, NUM\_LETTER)

return c - 'a';

}

struct Node {

Node() {init();}

Node \*son[NUM\_LETTER], \*father, \*suffix;

int ch;

bool isWord;

void init() {

for (int i = 0; i < NUM\_LETTER; ++i) son[i] = NULL;

father = suffix = NULL;

ch = 0;

isWord = false;

}

};

struct ACAutomaton {

int now;

Node v[MAX\_NODES], \*root;

void init() {

now = 1;

v[0].init();

root = &v[0];

root->father = root;

root->ch = -1;

}

void insert(char s[]) {

Node \*x = root;

int l = strlen(s);

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

int k = char2int(s[i]);

if (NULL == x->son[k]) {

v[now].init();

x->son[k] = &v[now];

v[now].father = x;

v[now].ch = k;

++now;

}

x = x->son[k];

}

x->isWord = true;

}

void build() {

queue<Node \*> Q;

root->suffix = root;

for (int i = 0; i < NUM\_LETTER; ++i) {

if (NULL == root->son[i])

root->son[i] = root;

else {

root->son[i]->suffix = root;

Q.push(root->son[i]);

}

}

while (!Q.empty()) {

Node \*x = Q.front();

Q.pop();

if (NULL == x->suffix)

x->suffix = x->father->suffix->son[x->ch];

x->isWord |= x->suffix->isWord;

for (int i = 0; i < NUM\_LETTER; ++i)

if (NULL == x->son[i])

x->son[i] = x->suffix->son[i];

else

Q.push(x->son[i]);

}

}

}acAutomaton;

char s[LEN\_PATTERN], text[LEN\_TEXT];

int main() {

int T = 1, N;

while (T--) {

acAutomaton.init();

scanf("%d", &N);

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

scanf(" %s", s), acAutomaton.insert(s);

acAutomaton.build();

scanf(" %s", text);

Node \*x = acAutomaton.root;

int l = strlen(text);

bool ans = false;

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

x = x->son[char2int(text[i])];

ans |= x->isWord;

}

printf("%s\n", ans ? "YES" : "NO");

}

return 0;

}

# LongestPalindromicSubstring

#include <cstdio>

#include <cstring>

const int MAXN = 2000005;

char s[MAXN];

int f[MAXN];

// s[i - f[i] .. i + f[i]] is the longest palindrome centered at i

int main() {

int T;

scanf("%d", &T);

while (T--) {

scanf(" %s", s);

int N = strlen(s);

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

s[(i << 1) + 1] = s[i], s[i << 1] = '$';

s[N << 1] = '$';

N = (N << 1) + 1;

int j = 0, ans = 0;

f[0] = 0;

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

if (j + j - i < 1 || f[j] + j <= i + f[j + j - i]) {

f[i] = f[j] + j - i;

if (f[i] < 0) f[i] = 0;

j = i;

while (i + f[i] + 1 < N && i - f[i] - 1 >= 0 && s[i + f[i] + 1] == s[i - f[i] - 1]) ++f[i];

}

else

f[i] = f[j + j - i];

if (f[i] > ans) ans = f[i];

}

printf("%d\n", ans);

}

return 0;

}

# GaussianElimination

#include <cstdio>

#include <algorithm>

using namespace std;

// N: number of variables; M: number of equations

const int MAXN = 505, MAXM = 1005;

const double EPS = 1e-18;

struct Gauss {

int N, M;

double a[MAXM][MAXN], b[MAXM];

// Solve for ax = b, where x is a column vector

enum {NO\_SOLUTION, MANY\_SOLUTION, UNIQUE\_SOLUTION};

bool doubleEq(double x, double y) {

return x - y <= EPS && y - x <= EPS;

}

bool isZeroRow(int x) {

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

if (!doubleEq(a[x][i], 0)) return false;

return true;

}

void swapRow(int x, int y) {

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

swap(a[x][i], a[y][i]);

swap(b[x], b[y]);

}

void addRow(int x, int y, double c) { // row(y) = row(y) + c \* row(x)

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

a[y][i] += a[x][i] \* c;

b[y] += b[x] \* c;

}

int solve() {

bool manySolutionFlag = false;

int i, k = 0;

for (i = 1; i <= N && k < N; ++i) {

int j;

bool flag = false;

while (k < N && !flag) {

++k;

for (j = i; j <= M; ++j) {

if (!doubleEq(a[j][k], 0)) {

if (j != i) swapRow(j, i);

flag = true;

break;

}

}

if (!flag) manySolutionFlag = true;

}

if (!flag) break;

for (j = 1; j <= M; ++j)

if (i != j) addRow(i, j, -a[j][k] / a[i][k]);

}

for (; i <= M; ++i)

if (isZeroRow(i) && !doubleEq(b[i], 0)) return NO\_SOLUTION;

if (manySolutionFlag) return MANY\_SOLUTION;

for (int i = N; i >= 1; --i)

b[i] /= a[i][i], a[i][i] = 1;

return UNIQUE\_SOLUTION;

}

}gauss;

int main() {

scanf("%d%d", &gauss.N, &gauss.M);

for (int i = 1; i <= gauss.M; ++i) {

for (int j = 1; j <= gauss.N; ++j)

scanf("%lf", &gauss.a[i][j]);

scanf("%lf", &gauss.b[i]);

}

int result = gauss.solve();

if (Gauss::NO\_SOLUTION == result) puts("No solutions");

else if (Gauss::MANY\_SOLUTION == result) puts("Many solutions");

else {

for (int i = 1; i <= gauss.N; ++i)

printf("%d\n", (int)(gauss.b[i] + 0.5));

}

return 0;

}

# FFT

#include <cstdio>

#include <cstring>

#include <cmath>

#include <algorithm>

using namespace std;

const double Pi = 3.14159265358979;

struct Complex {

double x, y; // x + yi

Complex(double x = 0, double y = 0): x(x), y(y) {}

Complex operator +(const Complex &B) {

return Complex(x + B.x, y + B.y);

}

Complex operator -(const Complex &B) {

return Complex(x - B.x, y - B.y);

}

Complex operator \*(const Complex &B) {

return Complex(x \* B.x - y \* B.y, x \* B.y + y \* B.x);

}

};

struct FFT {

void rearrange(Complex arr[], int len) { // len must be a power of 2

for (int i = 1, j = len >> 1; i < len - 1; ++i) {

if (i < j) swap(arr[i], arr[j]);

int k = len >> 1;

while (j >= k) j -= k, k >>= 1;

j += k;

}

}

void work(Complex y[], int len, int mode) {

rearrange(y, len);

double c = -mode \* 2 \* Pi;

for (int h = 2; h <= len; h <<= 1) {

Complex omegaN(cos(c / h), sin(c / h));

for (int j = 0, h2 = h >> 1; j < len; j += h) {

Complex omega(1, 0);

for (int k = j; k < j + h2; ++k) {

Complex a = y[k], b = omega \* y[k + h2];

y[k] = a + b;

y[k + h2] = a - b;

omega = omega \* omegaN;

}

}

}

if (mode == IDFT) {

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

y[i].x /= len, y[i].y /= len;

}

}

enum Mode{DFT = 1, IDFT = -1};

}fft;

const int MAXL = 50005;

char s1[MAXL], s2[MAXL];

Complex z1[MAXL << 2], z2 [MAXL << 2];

int ans[MAXL << 2];

// When you want to multiply two things, do DFT on both, multiply them, and do IDFT on the result

int main() {

while (scanf(" %s %s", s1, s2) != EOF) {

int len = 1, l1 = strlen(s1), l2 = strlen(s2);

while (len < l1 + l2) len <<= 1;

for (int i = 0; i < l1; ++i)

z1[i].x = s1[l1 - i - 1] - '0', z1[i].y = 0;

for (int i = l1; i < len; ++i)

z1[i].x = z1[i].y = 0;

for (int i = 0; i < l2; ++i)

z2[i].x = s2[l2 - i - 1] - '0', z2[i].y = 0;

for (int i = l2; i < len; ++i)

z2[i].x = z2[i].y = 0;

fft.work(z1, len, FFT::DFT);

fft.work(z2, len, FFT::DFT);

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

z1[i] = z1[i] \* z2[i];

fft.work(z1, len, FFT::IDFT);

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

ans[i] = (int)(z1[i].x + 0.5);

int c = 0;

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

ans[i] += c, c = ans[i] / 10, ans[i] %= 10;

c = len - 1;

while (c > 0 && ans[c] == 0) --c;

while (c >= 0) printf("%d", ans[c--]);

puts("");

}

return 0;

}

# NTT

#include <cstdio>

#include <algorithm>

using namespace std;

const int MOD = (479 << 21) + 1;

const int G = 3; // Primitive root

long long fastPow(long long a, long long b) {

long long ans = 1;

a %= MOD;

while (b) {

if (b & 1) ans = (ans \* a) % MOD;

b >>= 1;

a = (a \* a) % MOD;

}

return ans;

}

struct NumberTheoreticTransform {

void rearrange(long long arr[], int len) { // len must be a power of 2

for (int i = 1, j = len >> 1; i < len - 1; ++i) {

if (i < j) swap(arr[i], arr[j]);

int k = len >> 1;

while (j >= k) j -= k, k >>= 1;

j += k;

}

}

void work(long long y[], int len, int mode) {

rearrange(y, len);

for (int h = 2; h <= len; h <<= 1) {

long long omegaN = fastPow(G, (MOD - 1) / h);

if (mode == INTT) omegaN = fastPow(omegaN, MOD - 2);

for (int j = 0, h2 = h >> 1; j < len; j += h) {

long long omega = 1;

for (int k = j; k < j + h2; ++k) {

long long a = y[k], b = (omega \* y[k + h2]) % MOD;

y[k] = (a + b) % MOD;

y[k + h2] = ((a - b) % MOD + MOD) % MOD;

omega = (omega \* omegaN) % MOD;

}

}

}

if (mode == INTT) {

long long inv = fastPow(len, MOD - 2);

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

y[i] = (y[i] \* inv) % MOD;

}

}

enum Mode{NTT, INTT};

}ntt;

bool isRoot(long long x, long long y) { // Test if y is a primitive root of x. Usually x is MOD, and if true is returned, we set G to y.

long long p = y;

for (long long i = 1; i < x - 1; ++i) {

p = (p \* y) % x;

if (p == y) return false;

}

return true;

}

# PrimeAndPhiAndMu

#include <cstdio>

#include <algorithm>

using namespace std;

struct NumberTheory {

static const int MAXN = 100005;

bool isPrime[MAXN];

int primeCount, primeList[MAXN], phi[MAXN], mu[MAXN];

// primeCount: number of prime numbers in [1, MAXN]

// primeList: array of all the prime numbers

// phi: the Euler's totient function. phi[N] is the number of integers between [1, N - 1] that are coprmime to N.

// mu: Mobius function. mu[N] = 0 or pow(-1, number of prime factors of N).

// Computation of phi or mu be commented out if not needed

NumberTheory() {

isPrime[1] = false;

phi[1] = 0;

mu[1] = 1;

for (int i = 2; i < MAXN; ++i) isPrime[i] = true;

primeCount = 0;

sift();

}

void sift() {

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

if (isPrime[i])

primeList[primeCount++] = i, phi[i] = i - 1, mu[i] = -1;

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

if (i \* primeList[j] >= MAXN) break;

isPrime[i \* primeList[j]] = false;

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

phi[i \* primeList[j]] = phi[i] \* primeList[j];

mu[i \* primeList[j]] = 0;

break;

}

else {

phi[i \* primeList[j]] = phi[i] \* (primeList[j] - 1);

mu[i \* primeList[j]] = -mu[i];

}

}

}

}

}numberTheory;

int main() {

int T, a, b, c, d, k;

scanf("%d", &T);

for (int nCase = 1; nCase <= T; ++nCase) {

scanf("%d%d%d%d%d", &a, &b, &c, &d, &k);

if (b > d) swap(b, d);

if (k == 0) {

printf("Case %d: 0\n", nCase);

continue;

}

long long ans = 0, t = 0;

for (int i = 1; i \* k <= b; ++i)

ans += ((long long)(b / (i \* k))) \* (d / (i \* k)) \* numberTheory.mu[i],

t += ((long long)(b / (i \* k)) \* (b / (i \* k)) \* numberTheory.mu[i]);

printf("Case %d: %I64d\n", nCase, ans - (t >> 1));

}

return 0;

}

# MöbiusInversionFormula

# PolicyBasedDataStructure\_RBTree

#include <cassert>

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

#include <ext/pb\_ds/tree\_policy.hpp> // Including tree\_order\_statistics\_node\_update (seems unnecessary)

using namespace \_\_gnu\_pbds;

typedef tree<int, null\_type, std::less<int>, rb\_tree\_tag, tree\_order\_statistics\_node\_update> Set;

Set S;

int main() {

S.insert(3); S.insert(7); S.insert(5);

// find\_by\_order() returns an iterator to the k-th largest element (counting from zero)

assert(\*S.find\_by\_order(2) == 7);

assert(S.find\_by\_order(3) == S.end());

assert(S.find\_by\_order(4) == S.end());

// order\_of\_key() returns the number of items in a set that are strictly smaller than the given item

assert(S.order\_of\_key(6) == 2);

return 0;

}