-------- Handbook/Algorithms/Ad\_hoc/base\_conversions.h --------

string toBaseN(int num, int N) {

string converted = num ? "" : "0";

for(int div=abs(num); div; div /= N) {

int value = div % N;

converted = char(value > 9 ? value + 'A' - 10 : value + '0') + converted;

}

return converted;

}

-------- Handbook/Algorithms/Ad\_hoc/bit\_manipulation.h --------

#define turnOffLastBit(S) ((S) & (S - 1))

#define turnOnLastZero(S) ((S) | (S + 1))

#define turnOffLastConsecutiveBits(S) ((S) & (S + 1))

#define turnOnLastConsecutiveZeroes(S) ((S) | (S - 1))

int MSB(int x) {

if(!x) return 0;

int ans = 1;

while(x>>1) x>>=1, ans<<=1;

return ans;

}

-------- Handbook/Algorithms/Ad\_hoc/dates.h --------

int toJulian(int day, int month, int year) {

return 1461 \* (year + 4800 + (month - 14) / 12) / 4 + 367 \* (month - 2 -

(month - 14) / 12 \* 12) / 12 - 3 \* ((year + 4900 + (month - 14) / 12)

/ 100) / 4 + day - 32075;

}

void toGregorian(int julian, int &day, int &month, int &year) {

int x, n, i, j;

x = julian + 68569;

n = 4 \* x / 146097;

x -= (146097 \* n + 3) / 4;

i = (4000 \* (x + 1)) / 1461001;

x -= 1461 \* i / 4 - 31;

j = 80 \* x / 2447;

day = x - 2447 \* j / 80;

x = j / 11;

month = j + 2 - 12 \* x;

year = 100 \* (n - 49) + i + x;

}

bool isLeap(int year) { return (year%4 == 0 && year%100 != 0) || year%400 == 0; }

-------- Handbook/Algorithms/Ad\_hoc/longest\_increasing\_subsequence.h --------

#define STRICTLY\_INCREASING

vi LongestIncreasingSubsequence(vi v) {

vii best;

vi parent(v.size(), -1);

FOR(i, 0, v.size()) {

#ifdef STRICTLY\_INCREASING

ii item = ii(v[i], 0);

vii::iterator it = lower\_bound(best.begin(), best.end(), item);

item.second = i;

#else

ii item = ii(v[i], i);

vii::iterator it = upper\_bound(best.begin(), best.end(), item);

#endif

if (it == best.end()) {

parent[i] = (best.size() == 0 ? -1 : best.back().second);

best.pb(item);

} else {

parent[i] = parent[it->second];

\*it = item;

}

}

vi lis;

for(int i=best.back().second; i >= 0; i=parent[i])

lis.pb(v[i]);

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

return lis;

}

-------- Handbook/Algorithms/Ad\_hoc/maximum\_subarray.h --------

int maximumSubarray(int numbers[], int N) {

int maxSoFar = numbers[0], maxEndingHere = numbers[0];

FOR(i, 1, N) {

if(maxEndingHere < 0) maxEndingHere = numbers[i];

else maxEndingHere += numbers[i];

maxSoFar = max(maxEndingHere, maxSoFar);

}

return maxSoFar;

}

-------- Handbook/Algorithms/Ad\_hoc/range\_or.h --------

int rangeOR(int A, int B) {

int value = 0;

for(int i=1<<(sizeof(int)-1); i; i >>= 1) {

value <<= 1;

value += A/i&1 || B/i&1 || A/i != B/i;

}

return value;

}

-------- Handbook/Algorithms/Ad\_hoc/shunting\_yard.h --------

void output(string x) {

cout << x << " ";

}

string readToken() {

string t; int c;

while((c = cin.peek()) != EOF) {

if(isalpha(c) || isdigit(c)) t.pb((char)c), cin.get();

else if(t != "") return t;

else {cin.get(); if(!isspace(c)) {t.pb(c); return t;}}

} return "";

}

#define LEFT 0

#define RIGHT 1

#define isOp(x) (prec.find(x) != prec.end())

void shunting() {

string token;

stack<string> ops;

map<string, int> prec;

prec["\*"] = prec["/"] = prec["%"] = 5;

prec["+"] = prec["-"] = 4;

map<string, int> assoc; // default 0

while((token = readToken()) != "") {

if(isOp(token)) {

while(!ops.empty()

&& ((assoc[token] == LEFT && prec[token] <= prec[ops.top()])

|| prec[token] < prec[ops.top()]))

output(ops.top()), ops.pop();

ops.push(token);

} else if(token == "(") {

ops.push(token);

} else if(token == ")") {

while(!ops.empty() && ops.top() != "(")

output(ops.top()), ops.pop();

// ops.empty() || ops.top() != "(" ====> MISMATCH

ops.pop();

} else // numbers vars

output(token);

}

while(!ops.empty()) { // if ops.top() == ")" || ops.top() == "(" =======> MISMATCH

output(ops.top()), ops.pop();

}

}

-------- Handbook/Algorithms/Graphs/articulation\_points\_and\_bridges.h --------

//edges[from].back().backEdge = edges[aux].size() - 1; //add this to Graph.connect

//edges[aux].back().backEdge = edges[from].size() - 1; //at the end, inside the if

vi low2, num2, parent, strongPoints;

int counter2, root, rootChildren;

void dfs1(Graph &g, int v) {

low2[v] = num2[v] = counter2++;

FORC(g.edges[v], edge) {

if(num2[edge->to] == -1) {

parent[edge->to] = v;

if(v == root) rootChildren++;

dfs1(g, edge->to);

if(low2[edge->to] >= num2[v]) strongPoints[v] = true;

if(low2[edge->to] > num2[v]) edge->strong = g.edges[edge->to][edge->backEdge].strong = true;

low2[v] = min(low2[v], low2[edge->to]);

} else if(edge->to != parent[v])

low2[v] = min(low2[v], num2[edge->to]);

}

}

vi articulationPointsAndBridges(Graph &g /\*Undirected\*/) {

counter2 = 0;

num2 = vi(g.V, -1), low2 = vi(g.V, 0), parent = vi(g.V, -1), strongPoints = vi(g.V, 0);

FOR(i, 0, g.V)

if(num2[i] == -1) {

root = i, rootChildren = 0;

dfs1(g, i);

strongPoints[root] = rootChildren > 1;

}

return strongPoints;

}

-------- Handbook/Algorithms/Graphs/bellman\_ford.h --------

vi bellmanFord(Graph &g, int source, bool &negativeCycle) {

vi distanceTo(g.V, INF);

distanceTo[source] = 0;

FOR(i, 0, g.V-1)

FOR(j, 0, g.V)

FORC(g.edges[j], edge)

distanceTo[edge->to] = min(distanceTo[edge->to], distanceTo[j] + edge->weight);

//to detect negative weight cycles:

FOR(i, 0, g.V)

FORC(g.edges[i], edge)

if(distanceTo[edge->to] > distanceTo[i] + edge->weight)

negativeCycle = true;

return distanceTo;

}

-------- Handbook/Algorithms/Graphs/DAGs/shortest-longest\_path.h --------

vi shortestPath(Graph &g) {

vi order = topologicalSort(g);

vi distanceTo(g.V, 0);

FOR(i, 0, g.V) {

int cv = order[i];

FORC(g.edges[cv], edge) {

if(distanceTo[edge->to] == 0)

distanceTo[edge->to] = INF;

distanceTo[edge->to] = min(distanceTo[edge->to], edge->weight + distanceTo[cv]);

}

}

return distanceTo;

}

-------- Handbook/Algorithms/Graphs/DAGs/topological\_sort.h --------

vi topologicalSort(Graph &g) {

vi order, inDegree(g.V, 0);

FOR(i, 0, g.V)

FORC(g.edges[i], edge)

inDegree[edge->to]++;

FOR(i, 0, g.V)

if(inDegree[i] == 0)

order.pb(i);

FOR(i, 0, order.size())

FORC(g.edges[order[i]], edge)

if(--inDegree[edge->to] == 0)

order.pb(edge->to);

return order;

}

void dfs(Graph &g, int currentVertex, vi &order, vi &visited) {

visited[currentVertex] = true;

FORC(g.edges[currentVertex], edge)

if(!visited[edge->to])

dfs(g, edge->to, order, visited);

order.pb(currentVertex);

}

//Recursive version

vi topologicalSort2(Graph &g) {

vi order, visited(g.V, 0);

FOR(i, 0, g.V)

if(!visited[i])

dfs(g, i, order, visited);

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

return order;

}

-------- Handbook/Algorithms/Graphs/dijkstra.h --------

vi dijkstra(Graph &g, int src) {

vi dist(g.V, INF);

dist[src] = 0;

priority\_queue<ii, vii, greater<ii> > pq;

pq.push(ii(0, src));

while(!pq.empty()) {

int cv = pq.top().second;

int d = pq.top().first;

pq.pop();

if(d > dist[cv]) continue;

FORC(g.edges[cv], edge)

if(dist[edge->to] > dist[cv] + edge->weight) {

dist[edge->to] = dist[cv] + edge->weight;

pq.push(ii(dist[edge->to], edge->to));

}

}

return dist;

}

-------- Handbook/Algorithms/Graphs/edge\_property\_check.h --------

#define UNVISITED 0

#define EXPLORED 1 //visited but not completed

#define VISITED 2 //visited and completed

#define TREE 0 // Edge from explored to unvisited

#define BACK 1 // Edge that is part of a cycle (not including bidirectional edges). From explored to explored

#define FORWARD 2 // Edge from explored to visited

void dfs3(Graph &g, int cv, vi &parent, vi &state) {

state[cv] = EXPLORED;

FORC(g.edges[cv], edge)

if(state[edge->to] == UNVISITED) {

edge->type = TREE;

parent[edge->to] = cv;

dfs3(g, edge->to, parent, state);

} else if(state[edge->to] == EXPLORED)

edge->type = BACK; //if(edge->to == parent[cv]) //bidirectional

else if(state[edge->to] == VISITED)

edge->type = FORWARD;

state[cv] = VISITED;

}

void edgeProperties(Graph &g) {

vi state(g.V, UNVISITED), parent(g.V, 0);

FOR(i, 0, g.V)

if(state[i] == UNVISITED)

dfs3(g, i, parent, state);

}

-------- Handbook/Algorithms/Graphs/edmonds\_karp.h --------

int augment(Graph &g, int flow, vi &parent, int source, int cv, int minEdge) {

if(cv == source)

return minEdge;

if(parent[cv] != -1) {

flow = augment(g, flow, parent, source, parent[cv], min(minEdge, g.edges[parent[cv]][cv].weight));

g.edges[parent[cv]][cv].weight -= flow;

g.edges[cv][parent[cv]].weight += flow;

}

return flow;

}

//O(V\*E^2)

int maxFlow(Graph &g, int source, int sink) {

int mf = 0, flow = -1;

while(flow) {

vi distanceTo(g.V, INF);

distanceTo[source] = 0;

queue<int> q; q.push(source);

vi parent(g.V, -1);

while(!q.empty()) {

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

if(cv == sink) break;

FOR(i, 0, g.V)

if(g.edges[cv][i].weight > 0 && distanceTo[i] == INF)

distanceTo[i] = distanceTo[cv] + 1, q.push(i), parent[i] = cv;

}

mf += flow = augment(g, 0, parent, source, sink, INF);

}

return mf;

}

-------- Handbook/Algorithms/Graphs/eulerian\_path.h --------

void dfs2(Graph &g, list<int> &path, list<int>::iterator it, int cv) {

bool last = true;

FORC(g.edges[cv], edge) {

if(!edge->visited) {

last = false;

edge->visited = 1;

g.edges[edge->to][edge->backEdge].visited = 1;

/\*FORC(g.edges[edge->to], neighborEdge) {

if(neighborEdge->to == cv && !neighborEdge->visited) {

g.edges[edge->to][neighborEdge - g.edges[edge->to].begin()].visited = 1;

break;

}

}\*/

dfs2(g, path, path.insert(it, cv), edge->to);

}

}

if(last) path.insert(path.begin(), cv);

}

//At most two vertices can have odd degree

vi getEulerianPath(Graph &g/\*undirected\*/, int initial) {

list<int> path;

dfs2(g, path, path.begin(), initial);

vi p;

FORC(path, it)

p.pb(\*it);

return reverse(p.begin(), p.end()), p;

}

-------- Handbook/Algorithms/Graphs/floyd\_warshall.h --------

#define MAX\_V 400

void floydWarshall(Graph &g, int distance[MAX\_V][MAX\_V]) {

FOR(i, 0, g.V-1)

FOR(j, i, g.V)

distance[i][j] = distance[j][i] = INF\*(i != j);

FOR(i, 0, g.V)

FOR(j, 0, g.edges[i].size())

distance[i][g.edges[i][j].to] = g.edges[i][j].weight;

FOR(i, 0, g.V)

FOR(j, 0, g.V)

FOR(k, 0, g.V)

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

}

-------- Handbook/Algorithms/Graphs/lowest\_common\_ancestor.h --------

struct LCA {

vi order, height, index;

SparseTable \*st;

LCA(Graph &g, int root) {

index.assign(g.V, -1);

dfs(g, root, 0, index);

st = new SparseTable(height);

}

~LCA() { delete st; }

void dfs(Graph &g, int cv, int h, vi &index) {

index[cv] = order.size();

order.pb(cv), height.pb(h);

FORC(g.edges[cv], edge)

if(index[edge->to] == -1) {

dfs(g, edge->to, height.back() + edge->weight, index);

order.pb(cv), height.pb(h);

}

}

int query(int i, int j) { return order[st->query(index[i], index[j])]; }

int distance(int i, int j) { return height[index[i]] + height[index[j]] - 2\*(height[index[query(i, j)]]); }

};

-------- Handbook/Algorithms/Graphs/maximum\_bipartite\_matching.h --------

int augment(Graph &g, int cv, vi &match, vi &visited) {

if(visited[cv]) return 0;

visited[cv] = 1;

FORC(g.edges[cv], edge)

if(match[edge->to] == -1 || augment(g, match[edge->to], match, visited))

match[edge->to] = cv; return 1;

return 0;

}

//nodes in the left set must be nodes [0, left)

//g must be unweighted directed bipartite graph

int maxBipartiteMatching(Graph &g, int left) {

int MCBM = 0;

vi match(g.V, -1);

FOR(cv, 0, left) {

vi visited(left, 0);

MCBM += augment(g, cv, match, visited);

}

return MCBM;

}

-------- Handbook/Algorithms/Graphs/minimum\_spanning\_tree.h --------

int \*comparator1;

bool compare(int a, int b) { return comparator1[a] < comparator1[b]; }

vi kruskal(vii &edges, int weight[], int V) {

vi order(edges.size()), minTree;

UnionFindDS ds(V);

comparator1 = weight;

FOR(i, 0, order.size()) order[i] = i;

sort(order.begin(), order.end(), compare);

for(int i=0; i<int(edges.size()) && int(minTree.size()) < V - 1; i++)

if(!ds.connected(edges[order[i]].first, edges[order[i]].second)) {

ds.connect(edges[order[i]].first, edges[order[i]].second);

minTree.pb(order[i]);

}

return minTree;

}

Graph\* comparator2;

struct Compare { bool operator()(ii a, ii b) { return comparator2->edges[a.first][a.second].weight > comparator2->edges[b.first][b.second].weight;} };

vii prim(Graph &g) {

vi visited(g.V, 0);

visited[0] = 1;

vii tree; //list of edges in the MST

int visitedNodes = 1;

comparator2 = &g;

priority\_queue<ii, vector<ii>, Compare> pq;

int cv = 0;

while(visitedNodes != g.V) {

FORC(g.edges[cv], edge)

if(!visited[edge->to])

pq.push(ii(cv, edge - g.edges[cv].begin()));

ii nextEdge;

do {

nextEdge = pq.top();

pq.pop();

} while(visited[g.edges[nextEdge.first][nextEdge.second].to] && !pq.empty());

tree.pb(nextEdge);

cv = g.edges[nextEdge.first][nextEdge.second].to;

visitedNodes++;

visited[cv] = 1;

}

return tree;

}

-------- Handbook/Algorithms/Graphs/strongly\_connected\_components.h --------

vi low1, num1, components;

int counter1, SCCindex;

vector<bool> visited;

stack<int> S;

void dfs(Graph &g, int cv) {

low1[cv] = num1[cv] = counter1++;

S.push(cv);

visited[cv] = true;

FORC(g.edges[cv], edge) {

if(num1[edge->to] == -1)

dfs(g, edge->to);

if(visited[edge->to])

low1[cv] = min(low1[cv], low1[edge->to]);

}

if(low1[cv] == num1[cv]) {

int index = SCCindex++;

while(true) {

int v = S.top(); S.pop(); visited[v] = 0;

components[v] = index;

if (cv == v)

break;

}

}

}

vi stronglyConnectedComponents(Graph &g/\*directed\*/) {

counter1 = 0, SCCindex = 0;

visited = vector<bool>(g.V, 0);

num1 = vi(g.V, -1), low1 = vi(g.V, 0), components = vi(g.V, 0);

S = stack<int>();

FOR(i, 0, g.V)

if(num1[i] == -1)

dfs(g, i);

return components;

}

-------- Handbook/Algorithms/Graphs/tree\_hash.h --------

const int INIT = 191, P1 = 701, P2 = 34943;

int hs(vector<vi> &children, int root) {

int value = INIT;

vi sub;

FORC(children[root], it)

sub.pb(hs(children, \*it));

sort(sub.begin(), sub.end());

FORC(sub, it)

value = ((value \* P1) ^ \*it) % P2;

return value % P2;

}

-------- Handbook/Algorithms/Graphs/tree\_height\_for\_each\_root.h --------

int getLongestPathDown(Graph &g, int cv, vii &longestPathDown, vii &secondLongestPathDown, vi &parent) {

FORC(g.edges[cv], edge) {

if(edge->to != parent[cv]) {

parent[edge->to] = cv;

int pathDownLength = 1 + getLongestPathDown(g, edge->to, longestPathDown, secondLongestPathDown, parent);

if(pathDownLength > longestPathDown[cv].second) {

secondLongestPathDown[cv] = longestPathDown[cv];

longestPathDown[cv] = ii(edge->to, pathDownLength);

} else if(pathDownLength > secondLongestPathDown[cv].second) {

secondLongestPathDown[cv] = ii(edge->to, pathDownLength);

}

}

}

return longestPathDown[cv].second;

}

void getLongestPath(Graph &g /\*unrooted tree\*/, vii &longestPath) {

longestPath.assign(g.V, ii(-1, 0));

vii longestPathDown(g.V, ii(-1, 1)), secondLongestPathDown(g.V, ii(-1, 1)), secondLongestPath(g.V, ii(-1, 0));

vi parent(g.V, -1);

getLongestPathDown(g, 0, longestPathDown, secondLongestPathDown, parent);

queue<int> q;

q.push(0);

while(!q.empty()) {

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

FORC(g.edges[cv], edge)

if(edge->to != parent[cv])

q.push(edge->to);

if(parent[cv] == -1) {

longestPath[cv] = longestPathDown[cv];

secondLongestPath[cv] = secondLongestPathDown[cv];

} else {

ii longestPathThroughParent = ii(parent[cv], (longestPath[parent[cv]].first != cv ? longestPath[parent[cv]].second : secondLongestPath[parent[cv]].second)+1);

if(longestPathThroughParent.second >= longestPathDown[cv].second) {

longestPath[cv] = longestPathThroughParent;

secondLongestPath[cv] = longestPathDown[cv];

} else if(longestPathThroughParent.second >= secondLongestPathDown[cv].second) {

longestPath[cv] = longestPathDown[cv];

secondLongestPath[cv] = longestPathThroughParent;

} else {

longestPath[cv] = longestPathDown[cv];

secondLongestPath[cv] = secondLongestPathDown[cv];

}

}

}

}

-------- Handbook/Algorithms/Mathematics/binomial\_coefficients.h --------

//max n=61

int nCr(int n, int r) {

int res = 1;

FOR(i, 0, r) res = res\*(n-i)/(i+1);

return res;

}

#define MAXN 68

long long pascal[MAXN][MAXN];

void buildPascal() {

FOR(n, 0, MAXN)

FOR(r, 0, n+1)

pascal[n][r] = (r == 0 || r == n) ? 1 : pascal[n-1][r-1] + pascal[n-1][r];

}

-------- Handbook/Algorithms/Mathematics/catalan\_numbers.h --------

int fact(int n) {

return n ? n\*fact(n-1) : 1;

}

int nthCatalan(int n) {

return 2\*fact(n)/(pow(fact(n), 2)\*(n+1));

}

int nextCatalan(int n, int previous) {

return previous\*2\*(2\*n+1)/(n+1);

}

-------- Handbook/Algorithms/Mathematics/cycle\_finding.h --------

int f(int i) { return i; }

// x[i] = f(x[i-1])

ii floydCycleFinding(int x0) {

int tortoise = f(x0), hare = f(f(x0)); //Encontrar el primer xi = x2i

while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }

int mu = 0; hare = x0; //Encontrar mu usando el rango i

while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu++; }

int lambda = 1; hare = f(tortoise); //Encontrar lambda teniendo mu

while (tortoise != hare) { hare = f(hare); lambda++; }

return ii(mu, lambda);

}

-------- Handbook/Algorithms/Mathematics/euclid.h --------

// computes gcd(a,b)

int gcd(int a, int b) {

int tmp;

while(b){a%=b; tmp=a; a=b; b=tmp;}

return a;

}

// computes lcm(a,b)

int lcm(int a, int b) {

return a/gcd(a,b)\*b;

}

// returns d = gcd(a,b); finds x,y such that d = ax + by

int extended\_euclid(int a, int b, int &x, int &y) {

int xx = y = 0;

int yy = x = 1;

while (b) {

int q = a/b;

int t = b; b = a%b; a = t;

t = xx; xx = x-q\*xx; x = t;

t = yy; yy = y-q\*yy; y = t;

}

return a;

}

// finds all solutions to ax = b (mod n)

vi modular\_linear\_equation\_solver(int a, int b, int n) {

int x, y;

vi solutions;

int d = extended\_euclid(a, n, x, y);

if (!(b%d)) {

x = mod (x\*(b/d), n);

FOR(i, 0, d)

solutions.pb(mod(x + i\*(n/d), n));

}

return solutions;

}

// computes b such that ab = 1 (mod n), returns -1 on failure

int mod\_inverse(int a, int n) {

int x, y;

int d = extended\_euclid(a, n, x, y);

if (d > 1) return -1;

return mod(x,n);

}

-------- Handbook/Algorithms/Mathematics/extended\_euclidean.h --------

//d = ax + by

void extendedEuclid(int a, int b, int &x, int &y, int &d) {

if(!b) { x = 1; y = 0; d = a; return; }

extendedEuclid(b, a%b, x, y, d);

int x1 = y;

int y1 = x - (a/b)\*y;

x = x1, y = y1;

}

//r = ax + by

void extendedEuclidean (int a, int b, int &x, int &y, int &r) {

int r0 = a, r1 = b, s0 = 1, t0 = 0, s1 = 0, t1 = 1;

while(r1) {

int aux, q;

aux = r1;

q = r0 / r1;

r1 = r0 % r1;

r0 = aux;

aux = s1;

s1 = s0 - q \* s1;

s0 = aux;

aux = t1;

t1 = t0 - q \* t1;

t0 = aux;

}

if(r0 < 0) r0 = -r0, s0 = -s0, t0 = -t0;

r = r0, x = s0, y = t0;

}

const int MOD = 13;

long long multInverse(long long a, long long n) {

long long x, y, d;

extendedEuclid(a, n, x, y, d);

while (x < 0)

x += n;

return x % MOD;

}

-------- Handbook/Algorithms/Mathematics/factmod.h --------

int factmod (int n, int p) {

int res = 1;

while(n > 1) {

res = (res \* modpow (p-1, n/p, p)) % p;

for(int i = 2; i <= n%p; i++)

res = (res \* i) % p;

n /= p;

}

return res % p;

}

-------- Handbook/Algorithms/Mathematics/fast\_exponentiation.h --------

double fastPow(double a, int n) {

if(n == 0) return 1;

if(n == 1) return a;

double t = fastPow(a, n>>1);

return t\*t\*fastPow(a, n&1);

}

-------- Handbook/Algorithms/Mathematics/fibonacci.h --------

long long fibn(int n) { //max 91

double goldenRatio = (1+sqrt(5))/2;

return round((pow(goldenRatio, n+1) - pow(1-goldenRatio, n+1))/sqrt(5));

}

long long fib[92];

void buildFibonacci() {

fib[0] = fib[1] = 1;

for(int i=2; i<=100; i++) fib[i] = fib[i-2] + fib[i-1];

}

long long fibonacci(int n) {

Matrix m = CREATE(2, 2);

m[0][0] = 1, m[0][1] = 1, m[1][0] = 1, m[1][1] = 0;

Matrix fib0 = CREATE(2, 1);

fib0[0][0] = 1, fib0[1][0] = 1; //fib0 y fib1

Matrix r = multiply(pow(m, n), fib0);

return r[1][0];

}

-------- Handbook/Algorithms/Mathematics/matrices.h --------

typedef vector<vector<double> > Matrix;

#define EPS 1E-7

#define CREATE(R, C) Matrix(R, vector<double>(C));

Matrix identity(int n) {

Matrix m = CREATE(n, n);

FOR(i, 0, n)

m[i][i] = 1;

return m;

}

Matrix multiply(Matrix m, double k) {

FOR(i, 0, m.size())

FOR(j, 0, m[0].size())

m[i][j] \*= k;

return m;

}

Matrix multiply(Matrix m1, Matrix m2) {

Matrix result = CREATE(m1.size(), m2[0].size());

if(m1[0].size() != m2.size())

return result;

FOR(i, 0, result.size())

FOR(j, 0, result[0].size())

FOR(k, 0, m1[0].size())

result[i][j] += m1[i][k]\*m2[k][j];

return result;

}

Matrix pow(Matrix m, int exp) {

if(!exp) return identity(m.size());

if(exp == 1) return m;

Matrix result = identity(m.size());

while(exp) {

if(exp & 1) result = multiply(result, m);

m = multiply(m, m);

exp >>= 1;

}

return result;

}

//solves AX=B, output: A^-1 in A, X in B, returns det(A)

double gaussJordan(Matrix &a, Matrix &b) {

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

vi irow(n), icol(n), ipiv(n);

double det = 1;

FOR(i, 0, n) {

int pj = -1, pk = -1;

FOR(j, 0, n) if (!ipiv[j])

FOR(k, 0, n) if (!ipiv[k])

if (pj == -1 || abs(a[j][k]) > abs(a[pj][pk])) { pj = j; pk = k; }

if (abs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }

ipiv[pk]++;

swap(a[pj], a[pk]);

swap(b[pj], b[pk]);

if (pj != pk) det \*= -1;

irow[i] = pj;

icol[i] = pk;

double c = 1.0 / a[pk][pk];

det \*= a[pk][pk];

a[pk][pk] = 1.0;

FOR(p, 0, n) a[pk][p] \*= c;

FOR(p, 0, m) b[pk][p] \*= c;

FOR(p, 0, n) if (p != pk) {

c = a[p][pk];

a[p][pk] = 0;

FOR(q, 0, n) a[p][q] -= a[pk][q] \* c;

FOR(q, 0, m) b[p][q] -= b[pk][q] \* c;

}

}

for(int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {

FOR(k, 0, n) swap(a[k][irow[p]], a[k][icol[p]]);

}

return det;

}

//returns the rank of a

int rref(Matrix &a) {

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

int r = 0;

FOR(c, 0, m) {

int j = r;

FOR(i, r+1, n)

if (abs(a[i][c]) > abs(a[j][c])) j = i;

if (abs(a[j][c]) < EPS) continue;

swap(a[j], a[r]);

double s = 1.0 / a[r][c];

FOR(j, 0, m) a[r][j] \*= s;

FOR(i, 0, n) if (i != r) {

double t = a[i][c];

FOR(j, 0, m) a[i][j] -= t \* a[r][j];

}

r++;

}

return r;

}

-------- Handbook/Algorithms/Mathematics/modpow.h --------

int mod(int a, int b) {

return ((a%b)+b)%b;

}

int modpow(int base, int exp, int modulus) {

base = mod(base, modulus);

int result = 1;

while (exp) {

if (exp & 1) result = mod(result \* base, modulus);

base = mod(base \* base, modulus);

exp >>= 1;

}

return result;

}

-------- Handbook/Algorithms/Mathematics/nth\_permutation.h --------

string nthPermutation(string seq/\*sorted\*/, int permNum) {

if(!seq.length()) return "";

int f = fact(seq.length() - 1);

int q = permNum/f, r = permNum%f;

return seq[q] + nthPermutation(seq.substr(0, q) + seq.substr(q+1), r);

}

-------- Handbook/Algorithms/Mathematics/primes.h --------

#define SIZE 1000000

bitset<SIZE> sieve;

void buildSieve() {

sieve.set();

sieve[0] = sieve[1] = 0;

int root = sqrt(SIZE);

FOR(i, 2, SIZE)

if (sieve[i] && i <= root)

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

sieve[j] = 0;

}

vi primesList;

void buildPrimesList() {

if(!sieve[2])

buildSieve();

primesList.reserve(SIZE/log(SIZE));

FOR(i, 2, SIZE+1)

if(sieve[i])

primesList.pb(i);

}

vii primeFactorization(long long N) {

vii factors;

long long idx = 0, pf = primesList[0];

while(pf\*pf <= N) {

while(N%pf==0) {

N /= pf;

if(factors.size() && factors.back().first == pf)

factors.back().second++;

else

factors.pb(ii(pf, 1));

}

pf = primesList[++idx];

}

if(N!=1) factors.pb(ii(N, 1));

return factors;

}

void getDivisors(vii pf, long long d, int index, vi &div)

{

if (index == pf.size()) {

div.pb(d);

return;

}

for (int i = 0; i <= pf[index].second; i++) {

getDivisors(pf, d, index+1, div);

d \*= pf[index].first;

}

return;

}

vi divisors(int N) {

vii pf = primeFactorization(N);

vi div;

getDivisors(pf, 1ll, 0, div);

sort(div.begin(), div.end());

return div;

}

bool isPrime(long long n) {

if(n < 2) return false;

if(n == 2 || n == 3) return true;

if(!(n&1 && n%3)) return false;

long long sqrtN = sqrt(n)+1;

for(long long i = 6LL; i <= sqrtN; i += 6)

if(!(n%(i-1)) || !(n%(i+1))) return false;

return true;

}

-------- Handbook/Algorithms/Search/binary\_search.h --------

const int UPPERBOUND = 0, LOWERBOUND = 1, ANY = 2;

int binarySearch(int array[], int searchValue, int left, int right, int type = ANY) {

int leftBound = left, rightBound = right;

while(left <= right) {

int mid = (left+right)/1;

if(searchValue > array[mid]) left = mid+1;

else if (searchValue < array[mid]) right = mid-1;

else {

if(type == UPPERBOUND) {

if(mid == rightBound || array[mid+1] != array[mid])

return mid;

left = mid+1;

} else if(type == LOWERBOUND) {

if(mid == leftBound || array[mid-1] != array[mid])

return mid;

right = mid-1;

} else {

return mid;

}

}

}

return -1;

}

-------- Handbook/Algorithms/Sorting/mergesort.h --------

int merge(int array[], int low, int mid, int high) {

int inversions = 0;

int sorted[high-low+1];

int p1 = low, p2 = mid+1, psorted = 0;

while(p1 <= mid && p2 <= high) {

if(array[p1] <= array[p2])

sorted[psorted++] = array[p1++];

else {

sorted[psorted++] = array[p2++];

inversions += mid-p1+1;

}

}

while(p1 <= mid) sorted[psorted++] = array[p1++];

while(p2 <= high) sorted[psorted++] = array[p2++];

FOR(i, low, high+1) array[i] = sorted[i-low];

return inversions;

}

//returns the number of inversions

int mergeSort(int array[], int low, int high) {

if(low < high) {

int mid = (low + high)/2;

int inversions = mergeSort(array, low, mid) + mergeSort(array, mid+1, high);

return inversions + merge(array, low, mid, high);

}

return 0;

}

-------- Handbook/Algorithms/Sorting/quicksort.h --------

void quickSort(int arr[], int left, int right) {

int pivot = arr[(left+right)/2];

int i = left, j = right;

while(i <= j) {

while(arr[i] < pivot) i++;

while(arr[j] > pivot) j--;

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

}

if(left < j) quickSort(arr, left, j);

if(i < right) quickSort(arr, i, right);

}

-------- Handbook/Algorithms/Strings/edit\_distance.h --------

int editDistance(string A, string B) {

int n = A.length(), m = B.length();

int dist[n+1][m+1];

dist[0][0] = 0;

FOR(i, 1, n+1) dist[i][0] = i;

FOR(j, 1, m+1) dist[0][j] = j;

FOR(i, 1, n+1)

FOR(j, 1, m+1)

dist[i][j] = min(dist[i-1][j-1] + (A[i-1] != B[j-1]), min(dist[i-1][j] + 1, dist[i][j-1] + 1));

return dist[n][m];

}

-------- Handbook/Algorithms/Strings/longest\_common\_subsequence.h --------

string LCS(string a, string b) {

int n = a.length(), m = b.length();

int D[n][m];

char c[n][m];

FOR(i, 0, n)

FOR(j, 0, m)

if(a[i] == b[j]) {

D[i][j] = i&&j ? D[i-1][j-1] + 1 : 1;

c[i][j] = a[i];

}

else {

c[i][j] = (i ? D[i-1][j] : 0) >= (j ? D[i][j-1] : 0);

D[i][j] = max(i ? D[i-1][j] : 0, j ? D[i][j-1] : 0);

}

string lcs;

while(n-- && m--) {

if(c[n][m] == 0) n++;

else if(c[n][m] == 1) m++;

else lcs = c[n][m] + lcs;

}

return lcs;

}

-------- Handbook/Algorithms/Strings/string\_matching.h --------

vi buildTable(string& pattern) {

vi table(pattern.length());

int i = 0, j = -1, m = pattern.length();

table[0] = -1;

while(i < m) {

while(j >= 0 && pattern[i] != pattern[j]) j = table[j];

i++, j++;

table[i] = j;

}

return table;

}

vi find(string& text, string& pattern) {

vi matches;

int i = 0, j = 0, n = text.length(), m = pattern.length();

vi table = buildTable(pattern);

while(i < n) {

while(j >= 0 && text[i] != pattern[j]) j = table[j];

i++, j++;

if(j == m) {

matches.pb(i-j);

j = table[j];

}

}

return matches;

}

-------- Handbook/Algorithms/Strings/subsequence\_counter.h --------

// Regresa cuantas veces subseq es subsequence de seq

int subseqCounter(string seq, string subseq) {

int n = seq.length(), m = subseq.length();

vi sub(m, 0);

FOR(i, 0, n)

for(int j = m-1; j >= 0; j--)

if(seq[i] == subseq[j]) {

if(j == 0) sub[0]++;

else sub[j] += sub[j-1];

}

return sub[m-1];

}

-------- Handbook/Data\_Structures/balanced\_binary\_search\_tree.h --------

#define LCHILD(n) ((n)->parent->left == (n))

template< typename K, typename Compare = less<K> >

class SplayTree {

Compare compare;

struct Node {

Node \*left, \*right, \*parent;

K key;

Node(K k, Node \*p) : key(k), parent(p), left(0), right(0) {}

};

Node \*root;

void insert(Node \*node, K key) {

Node \*parent = find(node, key);

if(parent->key == key) return;

(compare(key, parent->key) ? parent->left : parent->right) = new Node(key, parent);

}

Node \* find(Node \*node, K key) {

if(key == node->key) { splay(node); return node; }

if(compare(key, node->key)) return node->left ? find(node->left, key) : node;

return node->right ? find(node->right, key) : node;

}

void erase(Node \*node, K key) {

node = find(node, key);

if(node->key != key) return;

if(node == root && !node->left && !node->right) {

root = 0;

delete node;

} else if(node->left && node->right) {

Node \*pred = node->left;

while(pred->right) pred = pred->right;

swap(node->key, pred->key);

if(pred != root) (LCHILD(pred) ? pred->parent->left : pred->parent->right) = pred->left ? pred->left : pred->right;

if(pred->left || pred->right) (pred->left ? pred->left : pred->right)->parent = pred->parent;

delete pred;

} else {

if(node == root) root = node->left ? node->left : node->right;

else (LCHILD(node) ? node->parent->left : node->parent->right) = node->left ? node->left : node->right;

if(node->left || node->right) (node->left ? node->left : node->right)->parent = node->parent;

delete node;

}

}

void leftRotate(Node \*parent) {

Node \*child = parent->right;

parent->right = child->left;

if(child->left) child->left->parent = parent;

child->parent = parent->parent;

if(!parent->parent) root = child;

else if(LCHILD(parent)) parent->parent->left = child;

else parent->parent->right = child;

child->left = parent;

parent->parent = child;

}

void rightRotate(Node \*parent) {

Node \*child = parent->left;

parent->left = child->right;

if(child->right) child->right->parent = parent;

child->parent = parent->parent;

if(!parent->parent) root = child;

else if(!LCHILD(parent)) parent->parent->right = child;

else parent->parent->left = child;

child->right = parent;

parent->parent = child;

}

void splay(Node \*node) {

while(root != node) {

if(node->parent->parent) {

if(LCHILD(node)) {

if(LCHILD(node->parent)) {

rightRotate(node->parent->parent);

rightRotate(node->parent);

} else {

rightRotate(node->parent);

leftRotate(node->parent);

}

} else {

if(LCHILD(node->parent)) {

leftRotate(node->parent);

rightRotate(node->parent);

} else {

leftRotate(node->parent->parent);

leftRotate(node->parent);

}

}

} else if(LCHILD(node)) {

rightRotate(node->parent);

} else {

leftRotate(node->parent);

}

}

}

void dealloc(Node \*node) { if(node->left) dealloc(node->left); if(node->right) dealloc(node->right); delete node; }

public:

SplayTree() : root(0) {}

~SplayTree() { if(root) dealloc(root); }

void insert(K key) { if(root) insert(root, key); else root = new Node(key, 0); }

void erase(K key) { if(root) erase(root, key); }

bool contains(K key) { return root && find(root, key)->key == key; }

};

-------- Handbook/Data\_Structures/binary\_heap.h --------

template <typename T>

struct Heap {

vector<T> tree;

int last;

Heap(int size) : last(1) { tree.assign(size+1, 0); }

void push(T n) {

tree[last++] = n;

for(int i=last-1; i != 1 && tree[i>>1] < tree[i]; i>>=1)

swap(tree[i], tree[i>>1]);

}

void pop() {

swap(tree[--last], tree[1]);

for(int i=1; ((i<<1) < last && tree[i] < tree[i<<1]) || ((i<<1)+1 < last && tree[i] < tree[(i<<1)+1]);) {

int k = ((i<<1) + ((i<<1)+1 < last && tree[(i<<1)+1] > tree[i<<1]));

swap(tree[i], tree[k]);

i=k;

}

}

int top() { return tree[1]; }

bool empty() { return last == 1; }

bool size() { return last - 1; }

};

-------- Handbook/Data\_Structures/fenwick\_tree.h --------

struct FenwickTree {

vi ft;

FenwickTree(int N) { ft.assign(N, 0); }

int query(int to) { int sum = 0; while(to) sum += ft[to], to -= to&-to; return sum; }

int query(int from, int to) { if(from > to) swap(to, from); return query(to) - query(from - 1); }

void add(int i, int value) { while(i < int(ft.size())) ft[i] += value, i += i&-i;}

};

struct FenwickTree2D {

vvi ft;

FenwickTree2D(int R, int C) { ft.assign(R, vi(C, 0)); }

int query(int r, int c) {

int sum = 0;

for(; r; r-=r&-r)

for(int j=c; j; j-=j&-j)

sum += ft[r][j];

return sum;

}

int query(int r, int c, int R, int C) { return query(R, C) - query(r-1, C) - query(R, c-1) + query(r-1, c-1); }

void update(int r, int c , int val) {

for(; r<int(ft.size()); r+=r&-r)

for(int j=c; j<int(ft.size()); j+=j&-j)

ft[r][j] += val;

}

};

-------- Handbook/Data\_Structures/Geometry/lines.h --------

struct Line {

double a, b, c;

Line() : a(0), b(0), c(0) {}

Line(Point p1, Point p2) {

if(abs(p1.x-p2.x) < EPS) {

a = 1.0; b = 0.0; c = -p1.x;

} else {

a = -(double)(p1.y-p2.y)/(p1.x-p2.x);

b = 1.0;

c = -(double)(a\*p1.x)-p1.y;

}

}

};

bool areParallel(Line l1, Line l2) {

return (abs(l1.a-l2.a) < EPS) && (abs(l1.b-l2.b) < EPS); }

bool areSame(Line l1, Line l2) {

return areParallel(l1, l2) && (abs(l1.c-l2.c) < EPS); }

bool areIntersect(Line l1, Line l2, Point &p) {

if (areParallel(l1, l2)) return false;

p.x = (l2.b \* l1.c - l1.b \* l2.c) / (l2.a \* l1.b - l1.a \* l2.b);

if (abs(l1.b) > EPS) p.y = -(l1.a \* p.x + l1.c);

else p.y = -(l2.a \* p.x + l2.c);

return true;

}

// Interseccion de AB con CD

// \* WARNING: Does not work for collinear line segments!

bool lineSegIntersect(Point a, Point b, Point c, Point d) {

double ucrossv1 = cross(toVec(a, b), toVec(a, c));

double ucrossv2 = cross(toVec(a, b), toVec(a, d));

if (ucrossv1 \* ucrossv2 > 0) return false;

double vcrossu1 = cross(toVec(c, d), toVec(c, a));

double vcrossu2 = cross(toVec(c, d), toVec(c, b));

return (vcrossu1 \* vcrossu2 <= 0);

}

// Calcula la distancia de un punto P a una recta AB, y guarda en C la inters

double distToLine(Point p, Point a, Point b, Point &c) {

Vec ap = toVec(a, p), ab = toVec(a, b);

double u = dot(ap, ab) / norm\_sq(ab);

c = translate(a, scale(ab, u));

return dist(p, c);

}

// Distancia a de P a segmento AB

double distToLineSegment(Point p, Point a, Point b, Point &c) {

Vec ap = toVec(a, p), ab = toVec(a, b);

double u = dot(ap, ab) / norm\_sq(ab);

if (u < 0.0) { c = a; return dist(p, a); }

if (u > 1.0) { c = b; return dist(p, b); }

return distToLine(p, a, b, c);

}

-------- Handbook/Data\_Structures/Geometry/point.h --------

const double PI = 2\*asin(1);

bool eq(double a, double b) { return fabs(a-b) < EPS; }

bool les(double a, double b) { return !eq(a, b) && a < b; }

struct Point {

double x, y, z;

Point() : x(0), y(0), z(0) {}

Point(double x, double y) : x(x), y(y), z(0) {}

Point(double x, double y, double z) : x(x), y(y), z(z) {}

bool operator <(const Point &p) const {

return les(x, p.x) || (eq(x, p.x) && les(y, p.y)) || (eq(x, p.x) && eq(y, p.y) && les(z, p.z));

}

bool operator==(const Point &p) {

return eq(x, p.x) && eq(y, p.y) && eq(z, p.z);

}

};

double DEG\_to\_RAD(double deg) {

return deg/180\*2\*asin(1);

}

double dist(Point p1, Point p2) {

return sqrt(pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2) + pow(p1.z-p2.z, 2)); }

Point rotate(Point p, double theta) {

double rad = DEG\_to\_RAD(theta);

return Point(p.x\*cos(rad) - p.y\*sin(rad),

p.x\*sin(rad) + p.y\*cos(rad));

}

double ANG(double rad) { return rad\*180/PI; }

double angulo(Point p) {

double d = atan(double(p.y)/p.x);

if(p.x < 0)

d += PI;

else if(p.y < 0)

d += 2\*PI;

return ANG(d);

}

-------- Handbook/Data\_Structures/Geometry/polygons.h --------

typedef vector<Point> Polygon;

ll cross(const Point &O, const Point &A, const Point &B) {

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

}

Polygon convexHull(Polygon &P) {

int n = P.size(), k = 0;

Polygon H(2\*n);

sort(P.begin(), P.end());

FOR(i, 0, n) {

while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;

H[k++] = P[i];

}

for (int i = n-2, t = k+1; i >= 0; i--) {

while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;

H[k++] = P[i];

}

H.resize(k);

return H;

}

// return area when Points are in cw or ccw, p[0] = p[n-1]

double area(const Polygon &P) {

double result = 0.0, x1, y1, x2, y2;

for (int i = 0; i < (int)P.size()-1; i++) {

x1 = P[i].x; x2 = P[i+1].x;

y1 = P[i].y; y2 = P[i+1].y;

result += (x1\*y2-x2\*y1);

}

return abs(result) / 2.0;

}

bool isConvex(const Polygon &P) {

int sz = (int)P.size();

if (sz <= 3) return false;

bool isLeft = ccw(P[0], P[1], P[2]);

for (int i = 1; i < sz-1; i++)

if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)

return false;

return true;

}

bool inPolygon (Point pt, const Polygon &P) {

if((int)P.size() == 0) return false;

double sum = 0;

for (int i = 0; i < (int)P.size()-1; i++) {

if (ccw(pt, P[i], P[i+1]))

sum += angle(P[i], pt, P[i+1]);

else sum -= angle(P[i], pt, P[i+1]); }

return abs(abs(sum) - 2\*PI) < EPS;

}

// tests whether or not a given polygon (in CW or CCW order) is simple

bool IsSimple(const Polygon &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 (lineSegIntersect(p[i], p[j], p[k], p[l]))

return false;

}

}

return true;

}

Point lineIntersectSeg(Point p, Point q, Point A, Point B) {

double a = B.y - A.y;

double b = A.x - B.x;

double c = B.x\*A.y - A.x\*B.y;

double u = abs(a\*p.x + b\*p.y + c);

double v = abs(a\*q.x + b\*q.y + c);

return Point((p.x\*v + q.x\*u) / (u+v), (p.y\*v + q.y\*u) / (u+v));

}

// cuts polygon Q along line AB

Polygon cutPolygon(Point a, Point b, const Polygon &Q) {

Polygon P;

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

double left1 = cross(toVec(a, b), toVec(a, Q[i+1])), left2 = 0;

if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));

if (left1 > -EPS) P.pb(Q[i]);

if (left1 \* left2 < -EPS)

P.pb(lineIntersectSeg(Q[i], Q[i+1], a, b));

}

if (!P.empty() && !(P.back() == P.front()))

P.pb(P.front());

return P;

}

-------- Handbook/Data\_Structures/Geometry/triangles.h --------

struct Triangle {

Point A, B, C;

Triangle() {}

Triangle(Point A, Point B, Point C) : A(A), B(B), C(C) {}

};

double perimeter(double a, double b, double c) { return a+b+c; }

// Heron's formula

double area(double a, double b, double c){

double s = perimeter(a, b, c)\*0.5;

return sqrt(s\*(s-a)\*(s-b)\*(s-c));

}

double rInCircle(double ab, double bc, double ca){

return area(ab, bc, ca) / (0.5 \* perimeter(ab, bc, ca)); }

double rInCircle(Point a, Point b, Point c) {

return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

bool inCircle(Point p1, Point p2, Point p3, Point &ctr, double &r) {

r = rInCircle(p1, p2, p3);

if(abs(r) < EPS) return false;

Line l1, l2;

double ratio = dist(p1, p2) / dist(p1, p3);

Point p = translate(p2, scale(toVec(p2, p3), ratio/(1+ratio)));

l1 = Line(p1, p);

ratio = dist(p2, p1) / dist(p2, p3);

l2 = Line(p2, p);

areIntersect(l1, l2, ctr);

return true;

}

Point circumcenter(Point A, Point B, Point C) {

double D = 2\*(A.x\*(B.y - C.y) + B.x\*(C.y - A.y) + C.x\*(A.y - B.y));

double AA = A.x\*A.x + A.y\*A.y, BB = B.x\*B.x + B.y\*B.y, CC = C.x\*C.x + C.y\*C.y;

return Point((AA\*(B.y - C.y) + BB\*(C.y - A.y) + CC\*(A.y - B.y)) / D, (AA\*(C.x - B.x) + BB\*(A.x - C.x) + CC\*(B.x - A.x)) / D);

}

-------- Handbook/Data\_Structures/Geometry/vectors.h --------

struct Vec {

double x, y, z;

Vec(double x, double y, double z) : x(x), y(y), z(z) {}

Vec() : x(0), y(0), z(0) {}

Vec(double x, double y) : x(x), y(y), z(0) {}

Vec(Point a, Point b) : x(b.x-a.x), y(b.y-a.y), z(b.z-a.z) {}

};

Vec toVec(Point a, Point b){

return Vec(a, b); }

Vec scale(Vec v, double s) {

return Vec(v.x\*s, v.y\*s, v.z\*s); }

Point translate(Point p, Vec v) {

return Point(p.x+v.x, p.y+v.y, p.z+v.z); }

double dot(Vec a, Vec b) {

return (a.x\*b.x + a.y\*b.y + a.z\*b.z); }

double norm\_sq(Vec v) {

return v.x\*v.x + v.y\*v.y + v.z\*v.z; }

//angle in radians

Vec rotate(Vec v, double angle) {

Matrix rotation = CREATE(2, 2);

rotation[0][0] = rotation[1][1] = cos(angle);

rotation[1][0] = sin(angle);

rotation[0][1] = -rotation[1][0];

Matrix vec = CREATE(2, 1);

vec[0][0] = v.x, vec[0][1] = v.y;

Matrix res = multiply(rotation, vec);

Vec result(res[0][0], res[0][1]);

return result;

}

double cross (Vec a, Vec b) { return a.x\*b.y - a.y\*b.x; }

// returns true if r is on the left side of line pq

bool ccw(Point p, Point q, Point r){

return cross(toVec(p, q), toVec(p, r)) > 0; }

bool collinear(Point p, Point q, Point r) {

return abs(cross(toVec(p, q), toVec(p, r))) < EPS; }

double angle(Point a, Point o, Point b) { // returns angle aob in rad

Vec oa = toVec(o, a), ob = toVec(o, b);

return acos(dot(oa, ob) / sqrt(norm\_sq(oa) \* norm\_sq(ob)));

}

-------- Handbook/Data\_Structures/interval\_tree.h --------

#define LCHILD(n) ((n)->parent->left == (n))

class IntervalTree {

struct Node {

Node \*left, \*right, \*parent;

set<int> intervals;

int key, area;

bool isLeaf;

void unLeaf(int k) {

isLeaf = 0, key = k;

left = new Node(this), right = new Node(this);

}

Node(Node \*p) : parent(p), isLeaf(1), area(0) {}

Node(int k, Node \*p) : parent(p), area(0) { unLeaf(k); }

};

Node \*root;

void insert(Node \*node, int key) {

Node \*parent = find(node, key);

if(parent->key == key) return;

(key < parent->key ? parent->left : parent->right)->unLeaf(key);

}

void insert(Node \*node, int interval, int a, int b, int imin, int imax) {

if(a <= imin && b >= imax) { node->area = imax-imin; node->intervals.insert(interval); return; }

if(a < node->key)

insert(node->left, interval, a, b, imin, node->key);

if(b > node->key)

insert(node->right, interval, a, b, node->key, imax);

if(node->intervals.size() == 0)

node->area = (node->left ? node->left->area : 0) + (node->right ? node->right->area : 0);

}

Node \* find(Node \*node, int key) {

if(key == node->key) { return node; }

if(key < node->key) return !node->left->isLeaf ? find(node->left, key) : node;

return !node->right->isLeaf ? find(node->right, key) : node;

}

void query(Node \*node, int a, int b, int imin, int imax, set<int> &result) {

if(!node) return;

result.insert(node->intervals.begin(), node->intervals.end());

if(a < node->key)

query(node->left, a, b, imin, node->key, result);

if(b >= node->key)

query(node->right, a, b, node->key, imax, result);

}

void erase(Node \*node, int interval, int a, int b, int imin, int imax) {

if(a <= imin && b >= imax) {

node->intervals.erase(interval);

if(node->intervals.size() == 0)

node->area = (node->left ? node->left->area : 0) + (node->right ? node->right->area : 0);

return;

}

if(a < node->key)

erase(node->left, interval, a, b, imin, node->key);

if(b > node->key)

erase(node->right, interval, a, b, node->key, imax);

if(node->intervals.size() == 0)

node->area = (node->left ? node->left->area : 0) + (node->right ? node->right->area : 0);

}

void dealloc(Node \*node) { if(node->left) dealloc(node->left); if(node->right) dealloc(node->right); delete node; }

public:

IntervalTree() : root(0) {}

~IntervalTree() { if(root) dealloc(root); }

void insert(int key) { if(root) insert(root, key); else root = new Node(key, 0); }

bool contains(int key) { return root && find(root, key)->key == key; }

void insert(int interval, int a, int b) { insert(a); insert(b+1); insert(root, interval, a, b+1, -INF, INF); }

set<int> query(int a, int b) { set<int> s; if(root) query(root, a, b, -INF, INF, s); return s; }

void erase(int interval, int a, int b) { erase(root, interval, a, b+1, -INF, INF); }

int getArea() { if(root) return root->area - 1; return 0; }

};

-------- Handbook/Data\_Structures/lists\_graph.h --------

struct Edge {

int to, weight;

int backEdge, strong, type, visited; //optional

Edge(int to, int weight = 1) : to(to), weight(weight), strong(0), visited(0) {}

};

struct Graph {

int V; bool undirected;

vector<vector<Edge> > edges;

Graph(int v, bool undirected) : V(v), undirected(undirected) { edges.assign(V, vector<Edge>()); }

void connect(int from, Edge edge) {

edges[from].pb(edge);

if(undirected) {

int aux = edge.to;

edge.to = from;

edges[aux].pb(edge);

edges[from].back().backEdge = edges[aux].size() - 1; //optional

edges[aux].back().backEdge = edges[from].size() - 1; //optional

}

}

};

-------- Handbook/Data\_Structures/matrix\_graph.h --------

struct MatrixEdge {

int weight;

MatrixEdge(int weight = 1) : weight(weight) { }

};

struct MatrixGraph {

int V; bool undirected;

vector<vector<Edge> > edges;

MatrixGraph(int v, bool undirected) : V(v), undirected(undirected) {

edges.assign(V, vector<Edge>(V, Edge(0)));

}

void connect(int from, int to, int weight = 1) {

edges[from][to].weight = weight;

if(undirected) edges[to][from].weight = weight;

}

};

-------- Handbook/Data\_Structures/segment\_tree.h --------

struct Node {

Node() {}

Node(int pos, int value) {}

int ans() { return 0; }

};

vi values;

Node operator+(Node &lNode, Node &rNode) {

Node merged;

//Merge operation

return merged;

}

struct SegmentTree {

vector<Node> tree;

Node query(int treeIndex, int L, int R, int from, int to) {

if(L >= from && R <= to) return tree[treeIndex];

Node left, right;

bool queryL = false, queryR = false;

if(from <= (L+R)/2) left = query(treeIndex\*2, L, (L+R)/2, from, to), queryL = true;

if(to >= (L+R)/2 + 1) right = query(treeIndex\*2 + 1, (L+R)/2 + 1, R, from, to), queryR = true;

if(!queryL) return right;

if(!queryR) return left;

return left + right;

}

void pointUpdate(int treeIndex, int L, int R, int index, Node &value) {

if (index > R || index < L) return;

if(L == R) {

tree[treeIndex] = value;

return;

}

pointUpdate(treeIndex\*2, L, (L+R)/2, index, value);

pointUpdate(treeIndex\*2 + 1, (L+R)/2 + 1, R, index, value);

tree[treeIndex] = tree[treeIndex\*2] + tree[treeIndex\*2 + 1];

}

void initialize(int treeIndex, int L, int R, int from, int to) {

if(L == R) {

tree[treeIndex] = Node(L, values[L]);

return;

}

initialize(treeIndex\*2, L, (L+R)/2, from, to);

initialize(treeIndex\*2 + 1, (L+R)/2+1, R, from, to);

tree[treeIndex] = tree[treeIndex\*2] + tree[treeIndex\*2 + 1];

}

SegmentTree(vi A) {

tree.clear();

int N = A.size();

tree.assign(2\*(1<<(int(log(N)/log(2))+1)), Node());

values = vi(A.begin(), A.end());

initialize(1, 0, N-1, 0, N-1);

}

void pointUpdate(int i, int k) { Node n(i, k); values[i] = k; pointUpdate(1, 0, values.size()-1, i, n); }

int query(int from, int to) { return query(1, 0, values.size()-1, from, to).ans(); }

};

-------- Handbook/Data\_Structures/sparse\_table.h --------

struct SparseTable {

vi A; vvi M;

int log2(int n) { int i=0; while(n >>= 1) i++; return i; }

SparseTable(vi arr) { //O(NlogN)

int N = arr.size();

A.assign(N, 0);

M.assign(N, vi(log2(N)+1));

int i, j;

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

M[i][0] = i, A[i] = arr[i];

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

for(i=0; i + (1<<j) - 1 < N; i++)

if(A[M[i][j - 1]] < A[M[i + (1 << (j - 1))][j - 1]])

M[i][j] = M[i][j - 1];

else

M[i][j] = M[i + (1 << (j - 1))][j - 1];

}

//returns the index of the minimum value

int query(int i, int j) {

if(i > j) swap(i, j);

int k = log2(j-i+1);

if(A[M[i][k]] < A[M[j-(1 << k)+1][k]])

return M[i][k];

return M[j-(1 << k)+1][k];

}

};

-------- Handbook/Data\_Structures/suffix\_array.h --------

#define MAX\_N 100010

int RA[MAX\_N], SA[MAX\_N], LCP[MAX\_N];

void countingSort(int k, char S[], int n) {

vi c(max(int(300), n), 0), tempSA(n);

int sum = 0, maxi = max(int(300), n);

FOR(i, 0, n) c[i+k<n ? RA[i+k]:0]++;

FOR(i, 0, maxi) {

sum += c[i];

c[i] = sum - c[i];

}

FOR(i, 0, n)

tempSA[c[SA[i]+k<n?RA[SA[i]+k]:0]++] = SA[i];

FOR(i, 0, n)

SA[i] = tempSA[i];

}

//S must end with a <=47 char

//FOR(i, 0, n)

// cout << S+SA[i] << ": " << LCP[i] << endl;

void buildSA(char S[], int n) {

vi tempRA(n);

FOR(i, 0, n)

RA[i] = S[i], SA[i] = i;

for(int k=1, r=0; k<n; k<<=1) {

countingSort(k, S, n);

countingSort(0, S, n);

tempRA[SA[0]] = r = 0;

FOR(i, 1, n)

tempRA[SA[i]] = (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;

FOR(i, 0, n)

RA[i] = tempRA[i];

if(RA[SA[n-1]] == n-1) break;

}

}

ii findPattern(char S[], int n, char P[], int m) {

int lo = 0, hi = n-1, mid;

while(lo < hi) {

mid = (lo + hi) / 2;

if(strncmp(S+SA[mid], P, m) >= 0) hi = mid;

else lo = mid+1;

}

if(strncmp(S+SA[lo], P, m) != 0) return ii(-1, -1);

ii bounds; bounds.first = lo;

lo = 0; hi = n-1; mid = lo;

while(lo < hi) {

mid = (lo + hi)/2;

if(strncmp(S+SA[mid], P, m) > 0) hi = mid;

else lo = mid+1;

}

if(strncmp(S+SA[hi], P, m) != 0) hi--;

bounds.second = hi;

return bounds;

}

//Amortized O(n)

//LCP[i] = longest common prefix between SA[i] and SA[i-1], LCP[0] = 0

void buildLCP(char S[], int n) {

vi phi(n), plcp(n);

int L = 0;

phi[SA[0]] = -1;

FOR(i, 1, n)

phi[SA[i]] = SA[i-1];

FOR(i, 0, n) {

if(phi[i] == -1) { plcp[i] = 0; continue; }

while(S[i+L] == S[phi[i]+L]) L++;

plcp[i] = L;

L = max(L-1, int(0));

}

FOR(i, 0, n) LCP[i] = plcp[SA[i]];

}

/\*

mint main() {

char S[7] = "ababc$";

int n = strlen(S);

buildSA(S, n);

buildLCP(S, n);

FOR(i, 0, n)

cout << i << " " << LCP[i] << " " << S+SA[i] << endl;

FOR(i, 1, n)

{

if(LCP[i])

{

int l = i-1;

while(LCP[l] >= LCP[i]) l--;

int j = l;

while(j<=i || (j<n && LCP[j] >= LCP[i])) j++;

int freq = j-l;

int len = LCP[i];

int startIndex = SA[i];

}

}

}

\*/

-------- Handbook/Data\_Structures/trie.h --------

#define ALPHABET\_SIZE 52

int getIndex(char c) {

if(c >= 'A' && c <= 'Z')

return c-'A';

return c-'a'+26;

}

struct Trie {

int words, prefixes;

Trie \*edges[ALPHABET\_SIZE];

Trie() : words(0), prefixes(0) { FOR(i, 0, ALPHABET\_SIZE) edges[i] = 0; }

~Trie(){ FOR(i, 0, ALPHABET\_SIZE) if(edges[i]) delete edges[i]; }

void insert(char \*word, int pos = 0) {

if(word[pos] == 0) {

words++;

return;

}

prefixes++;

int index = getIndex(word[pos]);

if(edges[index] == 0)

edges[index] = new Trie;

edges[index]->insert(word, pos+1);

}

int countWords(char \*word, int pos = 0) {

if(word[pos] == 0)

return words;

int index = getIndex(word[pos]);

if(edges[index]==0)

return 0;

return edges[index]->countWords(word, pos+1);

}

int countPrefix(char \*word, int pos = 0) {

if(word[pos] == 0)

return prefixes;

int index = getIndex(word[pos]);

if(edges[index] == 0)

return 0;

return edges[index]->countPrefix(word, pos+1);

}

};

-------- Handbook/Data\_Structures/union\_find\_disjoint\_sets.h --------

struct UnionFindDS {

vi tree;

UnionFindDS(int n) { FOR(i, 0, n) tree.pb(i); }

int root(int i) { return tree[i] == i ? i : tree[i] = root(tree[i]); }

bool connected(int i, int j) {return root(i) == root(j);}

void connect(int i, int j) { tree[root(i)] = tree[root(j)]; }

};

struct UnionFindDS2 {

vi tree, sizes;

int N;

UnionFindDS2(int n) : N(n) {

tree.reserve(n);

FOR(i, 0, n) tree[i] = i;

sizes.assign(n, 1);

}

int root(int i) { return (tree[i] == i) ? i : (tree[i] = root(tree[i]));}

int countSets() { return N;}

int getSize(int i) { return sizes[root(i)];}

bool connected(int i, int j) { return root(i) == root(j);}

void connect(int i, int j) {

int ri = root(i), rj = root(j);

if(ri != rj) {

N--;

sizes[rj] += sizes[ri];

tree[ri] = rj;

}

}

};

-------- Handbook/External/edmonds\_graph\_matching.cpp --------

struct edge {

int v, nx;

};

const int MAXN = 1000, MAXE = 2000;

edge graph[MAXE];

int last[MAXN], match[MAXN], px[MAXN], base[MAXN], N, edges;

bool used[MAXN], blossom[MAXN], lused[MAXN];

inline void add\_edge(int u, int v) {

graph[edges] = (edge) {v, last[u]};

last[u] = edges++;

graph[edges] = (edge) {u, last[v]};

last[v] = edges++;

}

void mark\_path(int v, int b, int children) {

while (base[v] != b) {

blossom[base[v]] = blossom[base[match[v]]] = true;

px[v] = children;

children = match[v];

v = px[match[v]];

}

}

int lca(int a, int b) {

memset(lused, 0, N);

while (1) {

lused[a = base[a]] = true;

if (match[a] == -1)

break;

a = px[match[a]];

}

while (1) {

b = base[b];

if (lused[b])

return b;

b = px[match[b]];

}

}

int find\_path(int root) {

memset(used, 0, N);

memset(px, -1, sizeof(int) \* N);

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

base[i] = i;

used[root] = true;

queue<int> q;

q.push(root);

int v, e, to, i;

while (!q.empty()) {

v = q.front(); q.pop();

for (e = last[v]; e >= 0; e = graph[e].nx) {

to = graph[e].v;

if (base[v] == base[to] || match[v] == to)

continue;

if (to == root || (match[to] != -1 && px[match[to]] != -1)) {

int curbase = lca(v, to);

memset(blossom, 0, N);

mark\_path(v, curbase, to);

mark\_path(to, curbase, v);

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

if (blossom[base[i]]) {

base[i] = curbase;

if (!used[i]) {

used[i] = true;

q.push(i);

}

}

} else if (px[to] == -1) {

px[to] = v;

if (match[to] == -1)

return to;

to = match[to];

used[to] = true;

q.push(to);

}

}

}

return -1;

}

void build\_pre\_matching() {

int u, e, v;

for (u = 0; u < N; ++u)

if (match[u] == -1)

for (e = last[u]; e >= 0; e = graph[e].nx) {

v = graph[e].v;

if (match[v] == -1) {

match[u] = v;

match[v] = u;

break;

}

}

}

void edmonds() {

memset(match, 0xff, sizeof(int) \* N);

build\_pre\_matching();

int i, v, pv, ppv;

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

if (match[i] == -1) {

v = find\_path(i);

while (v != -1) {

pv = px[v], ppv = match[pv];

match[v] = pv, match[pv] = v;

v = ppv;

}

}

}

-------- Handbook/External/roman\_numerals.h --------

string fill(char c, int n) {

string s;

while(n--) s += c;

return s;

}

string toRoman(int n) {

if( n < 4 ) return fill( 'i', n );

if( n < 6 ) return fill( 'i', 5 - n ) + "v";

if( n < 9 ) return string( "v" ) + fill( 'i', n - 5 );

if( n < 11 ) return fill( 'i', 10 - n ) + "x";

if( n < 40 ) return fill( 'x', n / 10 ) + toRoman( n % 10 );

if( n < 60 ) return fill( 'x', 5 - n / 10 ) + 'l' + toRoman( n % 10 );

if( n < 90 ) return string( "l" ) + fill( 'x', n / 10 - 5 ) + toRoman( n % 10 );

if( n < 110 ) return fill( 'x', 10 - n / 10 ) + "c" + toRoman( n % 10 );

if( n < 400 ) return fill( 'c', n / 100 ) + toRoman( n % 100 );

if( n < 600 ) return fill( 'c', 5 - n / 100 ) + 'd' + toRoman( n % 100 );

if( n < 900 ) return string( "d" ) + fill( 'c', n / 100 - 5 ) + toRoman( n % 100 );

if( n < 1100 ) return fill( 'c', 10 - n / 100 ) + "m" + toRoman( n % 100 );

if( n < 4000 ) return fill( 'm', n / 1000 ) + toRoman( n % 1000 );

return "?";

}

-------- Handbook/Utility/header.h --------

/\*

\*/

#include <algorithm>

#include <bitset>

#include <cmath>

#include <cstdio>

#include <cstring>

#include <deque>

#include <iomanip>

#include <iostream>

#include <queue>

#include <list>

#include <map>

#include <numeric>

#include <set>

#include <sstream>

#include <stack>

#include <utility>

#include <vector>

#include <cstdlib>

#define INF 1000000000

#define FOR(i, a, b) for(int i=int(a); i<int(b); i++)

#define FORC(cont, it) for(typeof((cont).begin()) it=(cont).begin(); it!=(cont).end(); it++)

#define pb push\_back

#define mp make\_pair

typedef int mint;

#define int ll

using namespace std; typedef long long ll; typedef pair<int, int> ii; typedef vector<int> vi; typedef vector<ii> vii; typedef vector<vi> vvi;

-------- Handbook/Utility/io.h --------

const int BUFFSIZE = 10240;

char BUFF[BUFFSIZE + 1], \*ppp = BUFF;

int RR, CHAR, SIGN, BYTES = 0;

#define GETCHAR(c) { \

if(ppp-BUFF==BYTES && (BYTES==0 || BYTES==BUFFSIZE)) { BYTES = fread(BUFF,1,BUFFSIZE,stdin); ppp=BUFF; } \

if(ppp-BUFF==BYTES && (BYTES>0 && BYTES<BUFFSIZE)) { BUFF[0] = 0; ppp=BUFF; } \

c = \*ppp++; \

}

#define DIGIT(c) (((c) >= '0') && ((c) <= '9'))

#define MINUS(c) ((c)== '-')

#define GETNUMBER(n) { \

n = 0; SIGN = 1; do { GETCHAR(CHAR); } while(!(DIGIT(CHAR) || MINUS(CHAR))); \

if(MINUS(CHAR)) { SIGN = -1; GETCHAR(CHAR); } \

while(DIGIT(CHAR)) { n = 10\*n + CHAR-'0'; GETCHAR(CHAR); } if(SIGN == -1) { n = -n; } \

}

-------- Handbook/Utility/notes.txt --------

Primes less than 1000:

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

-------- Handbook/Utility/run.sh --------

clear;g++ $1 -op&&python -c"import re;print(re.search(r'^/\\*\n((?s).\*)\\*/',open('$1').read()).group(1))"|./p;rm p

-------- Handbook/Utility/string\_number\_conversion.h --------

template <typename T>

string toString(T n) { ostringstream ss; ss << n; return ss.str(); }

template <typename T>

T toNum(const string &Text) { istringstream ss(Text); T result; return ss >> result ? result : 0; }

-------- Handbook/Utility/tricks.cpp --------

//tokenize a string

char str[100] = "jkans asjna asjnxa asmx", del[2] = " ";

for(char \*tk = strtok(str, del); tk; tk = strtok(0, del))

cout << tk << endl;

Bipartite graphs:

MVC = MCBM

MIS = V - MCBM

printf("%ld\n", strtol("222", 0, x)); //base x to long

regmatch\_t matches[1];

regcomp(&reg, pattern.c\_str(), REG\_EXTENDED|REG\_ICASE);

if(regexec(&reg, str.c\_str(), 1, matches, 0) == 0)

cout << "match" << endl;

regfree(&reg);

Dinic.cc 1/34

// Adjacency list implementation of Dinic's blocking flow algorithm.

// This is very fast in practice, and only loses to push-relabel flow.

//

// Running time:

// O(|V|^2 |E|)

//

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

//

// OUTPUT:

// - maximum flow value

// - To obtain the actual flow values, look at all edges with

// capacity > 0 (zero capacity edges are residual edges).

#include <cmath>

#include <vector>

#include <iostream>

#include <queue>

using namespace std;

const int INF = 2000000000;

struct Edge {

int from, to, cap, flow, index;

Edge(int from, int to, int cap, int flow, int index) :

from(from), to(to), cap(cap), flow(flow), index(index) {}

};

struct Dinic {

int N;

vector<vector<Edge> > G;

vector<Edge \*> dad;

vector<int> Q;

Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

void AddEdge(int from, int to, int cap) {

G[from].push\_back(Edge(from, to, cap, 0, G[to].size()));

if (from == to) G[from].back().index++;

G[to].push\_back(Edge(to, from, 0, 0, G[from].size() - 1));

}

long long BlockingFlow(int s, int t) {

fill(dad.begin(), dad.end(), (Edge \*) NULL);

dad[s] = &G[0][0] - 1;

int head = 0, tail = 0;

Q[tail++] = s;

while (head < tail) {

int x = Q[head++];

for (int i = 0; i < G[x].size(); i++) {

Edge &e = G[x][i];

if (!dad[e.to] && e.cap - e.flow > 0) {

dad[e.to] = &G[x][i];

Q[tail++] = e.to;

}

}

}

if (!dad[t]) return 0;

long long totflow = 0;

for (int i = 0; i < G[t].size(); i++) {

Edge \*start = &G[G[t][i].to][G[t][i].index];

int amt = INF;

for (Edge \*e = start; amt && e != dad[s]; e = dad[e->from]) {

if (!e) { amt = 0; break; }

amt = min(amt, e->cap - e->flow);

}

if (amt == 0) continue;

for (Edge \*e = start; amt && e != dad[s]; e = dad[e->from]) {

e->flow += amt;

G[e->to][e->index].flow -= amt;

}

totflow += amt;

}

return totflow;

}

long long GetMaxFlow(int s, int t) {

long long totflow = 0;

while (long long flow = BlockingFlow(s, t))

totflow += flow;

return totflow;

}

};

MinCostMaxFlow.cc 2/34

// Implementation of min cost max flow algorithm using adjacency

// matrix (Edmonds and Karp 1972). This implementation keeps track of

// forward and reverse edges separately (so you can set cap[i][j] !=

// cap[j][i]). For a regular max flow, set all edge costs to 0.

//

// Running time, O(|V|^2) cost per augmentation

// max flow: O(|V|^3) augmentations

// min cost max flow: O(|V|^4 \* MAX\_EDGE\_COST) augmentations

//

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

//

// OUTPUT:

// - (maximum flow value, minimum cost value)

// - To obtain the actual flow, look at positive values only.

#include <cmath>

#include <vector>

#include <iostream>

using namespace std;

typedef vector<int> VI;

typedef vector<VI> VVI;

typedef long long L;

typedef vector<L> VL;

typedef vector<VL> VVL;

typedef pair<int, int> PII;

typedef vector<PII> VPII;

const L INF = numeric\_limits<L>::max() / 4;

struct MinCostMaxFlow {

int N;

VVL cap, flow, cost;

VI found;

VL dist, pi, width;

VPII dad;

MinCostMaxFlow(int N) :

N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),

found(N), dist(N), pi(N), width(N), dad(N) {}

void AddEdge(int from, int to, L cap, L cost) {

this->cap[from][to] = cap;

this->cost[from][to] = cost;

}

void Relax(int s, int k, L cap, L cost, int dir) {

L val = dist[s] + pi[s] - pi[k] + cost;

if (cap && val < dist[k]) {

dist[k] = val;

dad[k] = make\_pair(s, dir);

width[k] = min(cap, width[s]);

}

}

L Dijkstra(int s, int t) {

fill(found.begin(), found.end(), false);

fill(dist.begin(), dist.end(), INF);

fill(width.begin(), width.end(), 0);

dist[s] = 0;

width[s] = INF;

while (s != -1) {

int best = -1;

found[s] = true;

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

if (found[k]) continue;

Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);

Relax(s, k, flow[k][s], -cost[k][s], -1);

if (best == -1 || dist[k] < dist[best]) best = k;

}

s = best;

}

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

pi[k] = min(pi[k] + dist[k], INF);

return width[t];

}

pair<L, L> GetMaxFlow(int s, int t) {

L totflow = 0, totcost = 0;

while (L amt = Dijkstra(s, t)) {

totflow += amt;

for (int x = t; x != s; x = dad[x].first) {

if (dad[x].second == 1) {

flow[dad[x].first][x] += amt;

totcost += amt \* cost[dad[x].first][x];

} else {

flow[x][dad[x].first] -= amt;

totcost -= amt \* cost[x][dad[x].first];

}

}

}

return make\_pair(totflow, totcost);

}

};

PushRelabel.cc 3/34

// Adjacency list implementation of FIFO push relabel maximum flow

// with the gap relabeling heuristic. This implementation is

// significantly faster than straight Ford-Fulkerson. It solves

// random problems with 10000 vertices and 1000000 edges in a few

// seconds, though it is possible to construct test cases that

// achieve the worst-case.

//

// Running time:

// O(|V|^3)

//

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

//

// OUTPUT:

// - maximum flow value

// - To obtain the actual flow values, look at all edges with

// capacity > 0 (zero capacity edges are residual edges).

#include <cmath>

#include <vector>

#include <iostream>

#include <queue>

using namespace std;

typedef long long LL;

struct Edge {

int from, to, cap, flow, index;

Edge(int from, int to, int cap, int flow, int index) :

from(from), to(to), cap(cap), flow(flow), index(index) {}

};

struct PushRelabel {

int N;

vector<vector<Edge> > G;

vector<LL> excess;

vector<int> dist, active, count;

queue<int> Q;

PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2\*N) {}

void AddEdge(int from, int to, int cap) {

G[from].push\_back(Edge(from, to, cap, 0, G[to].size()));

if (from == to) G[from].back().index++;

G[to].push\_back(Edge(to, from, 0, 0, G[from].size() - 1));

}

void Enqueue(int v) {

if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }

}

void Push(Edge &e) {

int amt = int(min(excess[e.from], LL(e.cap - e.flow)));

if (dist[e.from] <= dist[e.to] || amt == 0) return;

e.flow += amt;

G[e.to][e.index].flow -= amt;

excess[e.to] += amt;

excess[e.from] -= amt;

Enqueue(e.to);

}

void Gap(int k) {

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

if (dist[v] < k) continue;

count[dist[v]]--;

dist[v] = max(dist[v], N+1);

count[dist[v]]++;

Enqueue(v);

}

}

void Relabel(int v) {

count[dist[v]]--;

dist[v] = 2\*N;

for (int i = 0; i < G[v].size(); i++)

if (G[v][i].cap - G[v][i].flow > 0)

dist[v] = min(dist[v], dist[G[v][i].to] + 1);

count[dist[v]]++;

Enqueue(v);

}

void Discharge(int v) {

for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);

if (excess[v] > 0) {

if (count[dist[v]] == 1)

Gap(dist[v]);

else

Relabel(v);

}

}

LL GetMaxFlow(int s, int t) {

count[0] = N-1;

count[N] = 1;

dist[s] = N;

active[s] = active[t] = true;

for (int i = 0; i < G[s].size(); i++) {

excess[s] += G[s][i].cap;

Push(G[s][i]);

}

while (!Q.empty()) {

int v = Q.front();

Q.pop();

active[v] = false;

Discharge(v);

}

LL totflow = 0;

for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;

return totflow;

}

};

MinCostMatching.cc 4/34

///////////////////////////////////////////////////////////////////////////

// Min cost bipartite matching via shortest augmenting paths

//

// This is an O(n^3) implementation of a shortest augmenting path

// algorithm for finding min cost perfect matchings in dense

// graphs. In practice, it solves 1000x1000 problems in around 1

// second.

//

// cost[i][j] = cost for pairing left node i with right node j

// Lmate[i] = index of right node that left node i pairs with

// Rmate[j] = index of left node that right node j pairs with

//

// The values in cost[i][j] may be positive or negative. To perform

// maximization, simply negate the cost[][] matrix.

///////////////////////////////////////////////////////////////////////////

#include <algorithm>

#include <cstdio>

#include <cmath>

#include <vector>

using namespace std;

typedef vector<double> VD;

typedef vector<VD> VVD;

typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {

int n = int(cost.size());

// construct dual feasible solution

VD u(n);

VD v(n);

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

u[i] = cost[i][0];

for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);

}

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

v[j] = cost[0][j] - u[0];

for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);

}

// construct primal solution satisfying complementary slackness

Lmate = VI(n, -1);

Rmate = VI(n, -1);

int mated = 0;

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

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

if (Rmate[j] != -1) continue;

if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {

Lmate[i] = j;

Rmate[j] = i;

mated++;

break;

}

}

}

VD dist(n);

VI dad(n);

VI seen(n);

// repeat until primal solution is feasible

while (mated < n) {

// find an unmatched left node

int s = 0;

while (Lmate[s] != -1) s++;

// initialize Dijkstra

fill(dad.begin(), dad.end(), -1);

fill(seen.begin(), seen.end(), 0);

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

dist[k] = cost[s][k] - u[s] - v[k];

int j = 0;

while (true) {

// find closest

j = -1;

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

if (seen[k]) continue;

if (j == -1 || dist[k] < dist[j]) j = k;

}

seen[j] = 1;

// termination condition

if (Rmate[j] == -1) break;

// relax neighbors

const int i = Rmate[j];

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

if (seen[k]) continue;

const double new\_dist = dist[j] + cost[i][k] - u[i] - v[k];

if (dist[k] > new\_dist) {

dist[k] = new\_dist;

dad[k] = j;

}

}

}

// update dual variables

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

if (k == j || !seen[k]) continue;

const int i = Rmate[k];

v[k] += dist[k] - dist[j];

u[i] -= dist[k] - dist[j];

}

u[s] += dist[j];

// augment along path

while (dad[j] >= 0) {

const int d = dad[j];

Rmate[j] = Rmate[d];

Lmate[Rmate[j]] = j;

j = d;

}

Rmate[j] = s;

Lmate[s] = j;

mated++;

}

double value = 0;

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

value += cost[i][Lmate[i]];

return value;

}

MinCut.cc 6/34

// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.

//

// Running time:

// O(|V|^3)

//

// INPUT:

// - graph, constructed using AddEdge()

//

// OUTPUT:

// - (min cut value, nodes in half of min cut)

#include <cmath>

#include <vector>

#include <iostream>

using namespace std;

typedef vector<int> VI;

typedef vector<VI> VVI;

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {

int N = weights.size();

VI used(N), cut, best\_cut;

int best\_weight = -1;

for (int phase = N-1; phase >= 0; phase--) {

VI w = weights[0];

VI added = used;

int prev, last = 0;

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

prev = last;

last = -1;

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

if (!added[j] && (last == -1 || w[j] > w[last])) last = j;

if (i == phase-1) {

for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];

for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];

used[last] = true;

cut.push\_back(last);

if (best\_weight == -1 || w[last] < best\_weight) {

best\_cut = cut;

best\_weight = w[last];

}

} else {

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

w[j] += weights[last][j];

added[last] = true;

}

}

}

return make\_pair(best\_weight, best\_cut);

}

GraphCutInference.cc 7/34

// Special-purpose {0,1} combinatorial optimization solver for

// problems of the following by a reduction to graph cuts:

//

// minimize sum\_i psi\_i(x[i])

// x[1]...x[n] in {0,1} + sum\_{i < j} phi\_{ij}(x[i], x[j])

//

// where

// psi\_i : {0, 1} --> R

// phi\_{ij} : {0, 1} x {0, 1} --> R

//

// such that

// phi\_{ij}(0,0) + phi\_{ij}(1,1) <= phi\_{ij}(0,1) + phi\_{ij}(1,0) (\*)

//

// This can also be used to solve maximization problems where the

// direction of the inequality in (\*) is reversed.

//

// INPUT: phi -- a matrix such that phi[i][j][u][v] = phi\_{ij}(u, v)

// psi -- a matrix such that psi[i][u] = psi\_i(u)

// x -- a vector where the optimal solution will be stored

//

// OUTPUT: value of the optimal solution

//

// To use this code, create a GraphCutInference object, and call the

// DoInference() method. To perform maximization instead of minimization,

// ensure that #define MAXIMIZATION is enabled.

#include <vector>

#include <iostream>

using namespace std;

typedef vector<int> VI;

typedef vector<VI> VVI;

typedef vector<VVI> VVVI;

typedef vector<VVVI> VVVVI;

const int INF = 1000000000;

// comment out following line for minimization

#define MAXIMIZATION

struct GraphCutInference {

int N;

VVI cap, flow;

VI reached;

int Augment(int s, int t, int a) {

reached[s] = 1;

if (s == t) return a;

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

if (reached[k]) continue;

if (int aa = min(a, cap[s][k] - flow[s][k])) {

if (int b = Augment(k, t, aa)) {

flow[s][k] += b;

flow[k][s] -= b;

return b;

}

}

}

return 0;

}

int GetMaxFlow(int s, int t) {

N = cap.size();

flow = VVI(N, VI(N));

reached = VI(N);

int totflow = 0;

while (int amt = Augment(s, t, INF)) {

totflow += amt;

fill(reached.begin(), reached.end(), 0);

}

return totflow;

}

int DoInference(const VVVVI &phi, const VVI &psi, VI &x) {

int M = phi.size();

cap = VVI(M+2, VI(M+2));

VI b(M);

int c = 0;

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

b[i] += psi[i][1] - psi[i][0];

c += psi[i][0];

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

b[i] += phi[i][j][1][1] - phi[i][j][0][1];

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

cap[i][j] = phi[i][j][0][1] + phi[i][j][1][0] - phi[i][j][0][0] - phi[i][j][1][1];

b[i] += phi[i][j][1][0] - phi[i][j][0][0];

c += phi[i][j][0][0];

}

}

#ifdef MAXIMIZATION

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

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

cap[i][j] \*= -1;

b[i] \*= -1;

}

c \*= -1;

#endif

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

if (b[i] >= 0) {

cap[M][i] = b[i];

} else {

cap[i][M+1] = -b[i];

c += b[i];

}

}

int score = GetMaxFlow(M, M+1);

fill(reached.begin(), reached.end(), 0);

Augment(M, M+1, INF);

x = VI(M);

for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;

score += c;

#ifdef MAXIMIZATION

score \*= -1;

#endif

return score;

}

};

int main() {

// solver for "Cat vs. Dog" from NWERC 2008

int numcases;

cin >> numcases;

for (int caseno = 0; caseno < numcases; caseno++) {

int c, d, v;

cin >> c >> d >> v;

VVVVI phi(c+d, VVVI(c+d, VVI(2, VI(2))));

VVI psi(c+d, VI(2));

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

char p, q;

int u, v;

cin >> p >> u >> q >> v;

u--; v--;

if (p == 'C') {

phi[u][c+v][0][0]++;

phi[c+v][u][0][0]++;

} else {

phi[v][c+u][1][1]++;

phi[c+u][v][1][1]++;

}

}

GraphCutInference graph;

VI x;

cout << graph.DoInference(phi, psi, x) << endl;

}

return 0;

}

Geometry.cc 9/34

// C++ routines for computational geometry.

#include <iostream>

#include <vector>

#include <cmath>

#include <cassert>

using namespace std;

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 << ")";

}

// 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=c-d; 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 == l || 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, 1.166666)

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;

}

JavaGeometry.java 10/34

// In this example, we read an input file containing three lines, each

// containing an even number of doubles, separated by commas. The first two

// lines represent the coordinates of two polygons, given in counterclockwise

// (or clockwise) order, which we will call "A" and "B". The last line

// contains a list of points, p[1], p[2], ...

//

// Our goal is to determine:

// (1) whether B - A is a single closed shape (as opposed to multiple shapes)

// (2) the area of B - A

// (3) whether each p[i] is in the interior of B - A

//

// INPUT:

// 0 0 10 0 0 10

// 0 0 10 10 10 0

// 8 6

// 5 1

//

// OUTPUT:

// The area is singular.

// The area is 25.0

// Point belongs to the area.

// Point does not belong to the area.

import java.util.\*;

import java.awt.geom.\*;

import java.io.\*;

public class JavaGeometry {

// make an array of doubles from a string

static double[] readPoints(String s) {

String[] arr = s.trim().split("\\s++");

double[] ret = new double[arr.length];

for (int i = 0; i < arr.length; i++) ret[i] = Double.parseDouble(arr[i]);

return ret;

}

// make an Area object from the coordinates of a polygon

static Area makeArea(double[] pts) {

Path2D.Double p = new Path2D.Double();

p.moveTo(pts[0], pts[1]);

for (int i = 2; i < pts.length; i += 2) p.lineTo(pts[i], pts[i+1]);

p.closePath();

return new Area(p);

}

// compute area of polygon

static double computePolygonArea(ArrayList<Point2D.Double> points) {

Point2D.Double[] pts = points.toArray(new Point2D.Double[points.size()]);

double area = 0;

for (int i = 0; i < pts.length; i++){

int j = (i+1) % pts.length;

area += pts[i].x \* pts[j].y - pts[j].x \* pts[i].y;

}

return Math.abs(area)/2;

}

// compute the area of an Area object containing several disjoint polygons

static double computeArea(Area area) {

double totArea = 0;

PathIterator iter = area.getPathIterator(null);

ArrayList<Point2D.Double> points = new ArrayList<Point2D.Double>();

while (!iter.isDone()) {

double[] buffer = new double[6];

switch (iter.currentSegment(buffer)) {

case PathIterator.SEG\_MOVETO:

case PathIterator.SEG\_LINETO:

points.add(new Point2D.Double(buffer[0], buffer[1]));

break;

case PathIterator.SEG\_CLOSE:

totArea += computePolygonArea(points);

points.clear();

break;

}

iter.next();

}

return totArea;

}

// notice that the main() throws an Exception -- necessary to

// avoid wrapping the Scanner object for file reading in a

// try { ... } catch block.

public static void main(String args[]) throws Exception {

Scanner scanner = new Scanner(new File("input.txt"));

// also,

// Scanner scanner = new Scanner (System.in);

double[] pointsA = readPoints(scanner.nextLine());

double[] pointsB = readPoints(scanner.nextLine());

Area areaA = makeArea(pointsA);

Area areaB = makeArea(pointsB);

areaB.subtract(areaA);

// also,

// areaB.exclusiveOr (areaA);

// areaB.add (areaA);

// areaB.intersect (areaA);

// (1) determine whether B - A is a single closed shape (as

// opposed to multiple shapes)

boolean isSingle = areaB.isSingular();

// also,

// areaB.isEmpty();

if (isSingle)

System.out.println("The area is singular.");

else

System.out.println("The area is not singular.");

// (2) compute the area of B - A

System.out.println("The area is " + computeArea(areaB) + ".");

// (3) determine whether each p[i] is in the interior of B - A

while (scanner.hasNextDouble()) {

double x = scanner.nextDouble();

assert(scanner.hasNextDouble());

double y = scanner.nextDouble();

if (areaB.contains(x,y)) {

System.out.println ("Point belongs to the area.");

} else {

System.out.println ("Point does not belong to the area.");

}

}

// Finally, some useful things we didn't use in this example:

//

// Ellipse2D.Double ellipse = new Ellipse2D.Double (double x, double y,

// double w, double h);

//

// creates an ellipse inscribed in box with bottom-left corner (x,y)

// and upper-right corner (x+y,w+h)

//

// Rectangle2D.Double rect = new Rectangle2D.Double (double x, double y,

// double w, double h);

//

// creates a box with bottom-left corner (x,y) and upper-right

// corner (x+y,w+h)

//

// Each of these can be embedded in an Area object (e.g., new Area (rect)).

}

}

Geom3D.java 11/34

public class Geom3D {

// distance from point (x, y, z) to plane aX + bY + cZ + d = 0

public static double ptPlaneDist(double x, double y, double z,

double a, double b, double c, double d) {

return Math.abs(a\*x + b\*y + c\*z + d) / Math.sqrt(a\*a + b\*b + c\*c);

}

// distance between parallel planes aX + bY + cZ + d1 = 0 and

// aX + bY + cZ + d2 = 0

public static double planePlaneDist(double a, double b, double c,

double d1, double d2) {

return Math.abs(d1 - d2) / Math.sqrt(a\*a + b\*b + c\*c);

}

// distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)

// (or ray, or segment; in the case of the ray, the endpoint is the

// first point)

public static final int LINE = 0;

public static final int SEGMENT = 1;

public static final int RAY = 2;

public static double ptLineDistSq(double x1, double y1, double z1,

double x2, double y2, double z2, double px, double py, double pz,

int type) {

double pd2 = (x1-x2)\*(x1-x2) + (y1-y2)\*(y1-y2) + (z1-z2)\*(z1-z2);

double x, y, z;

if (pd2 == 0) {

x = x1;

y = y1;

z = z1;

} else {

double u = ((px-x1)\*(x2-x1) + (py-y1)\*(y2-y1) + (pz-z1)\*(z2-z1)) / pd2;

x = x1 + u \* (x2 - x1);

y = y1 + u \* (y2 - y1);

z = z1 + u \* (z2 - z1);

if (type != LINE && u < 0) {

x = x1;

y = y1;

z = z1;

}

if (type == SEGMENT && u > 1.0) {

x = x2;

y = y2;

z = z2;

}

}

return (x-px)\*(x-px) + (y-py)\*(y-py) + (z-pz)\*(z-pz);

}

public static double ptLineDist(double x1, double y1, double z1,

double x2, double y2, double z2, double px, double py, double pz,

int type) {

return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz, type));

}

}

Delaunay.cc 12/34

// 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;

}

Euclid.cc 13/34

// This is a collection of useful code for solving problems that

// involve modular linear equations. Note that all of the

// algorithms described here work on nonnegative integers.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

typedef vector<int> VI;

typedef pair<int,int> PII;

// return a % b (positive value)

int mod(int a, int b) {

return ((a%b)+b)%b;

}

// computes gcd(a,b)

int gcd(int a, int b) {

int tmp;

while(b){a%=b; tmp=a; a=b; b=tmp;}

return a;

}

// computes lcm(a,b)

int lcm(int a, int b) {

return a/gcd(a,b)\*b;

}

// returns d = gcd(a,b); finds x,y such that d = ax + by

int extended\_euclid(int a, int b, int &x, int &y) {

int xx = y = 0;

int yy = x = 1;

while (b) {

int q = a/b;

int t = b; b = a%b; a = t;

t = xx; xx = x-q\*xx; x = t;

t = yy; yy = y-q\*yy; y = t;

}

return a;

}

// finds all solutions to ax = b (mod n)

VI modular\_linear\_equation\_solver(int a, int b, int n) {

int x, y;

VI solutions;

int d = extended\_euclid(a, n, x, y);

if (!(b%d)) {

x = mod (x\*(b/d), n);

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

solutions.push\_back(mod(x + i\*(n/d), n));

}

return solutions;

}

// computes b such that ab = 1 (mod n), returns -1 on failure

int mod\_inverse(int a, int n) {

int x, y;

int d = extended\_euclid(a, n, x, y);

if (d > 1) return -1;

return mod(x,n);

}

// Chinese remainder theorem (special case): find z such that

// z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).

// Return (z,M). On failure, M = -1.

PII chinese\_remainder\_theorem(int x, int a, int y, int b) {

int s, t;

int d = extended\_euclid(x, y, s, t);

if (a%d != b%d) return make\_pair(0, -1);

return make\_pair(mod(s\*b\*x+t\*a\*y,x\*y)/d, x\*y/d);

}

// Chinese remainder theorem: find z such that

// z % x[i] = a[i] for all i. Note that the solution is

// unique modulo M = lcm\_i (x[i]). Return (z,M). On

// failure, M = -1. Note that we do not require the a[i]'s

// to be relatively prime.

PII chinese\_remainder\_theorem(const VI &x, const VI &a) {

PII ret = make\_pair(a[0], x[0]);

for (int i = 1; i < x.size(); i++) {

ret = chinese\_remainder\_theorem(ret.second, ret.first, x[i], a[i]);

if (ret.second == -1) break;

}

return ret;

}

int main() {

// expected: 2

cout << gcd(14, 30) << endl;

// expected: 2 -2 1

int x, y;

int d = extended\_euclid(14, 30, x, y);

cout << d << " " << x << " " << y << endl;

// expected: 95 45

VI sols = modular\_linear\_equation\_solver(14, 30, 100);

for (int i = 0; i < (int) sols.size(); i++) cout << sols[i] << " ";

cout << endl;

// expected: 8

cout << mod\_inverse(8, 9) << endl;

// expected: 23 56

// 11 12

int xs[] = {3, 5, 7, 4, 6};

int as[] = {2, 3, 2, 3, 5};

PII ret = chinese\_remainder\_theorem(VI (xs, xs+3), VI(as, as+3));

cout << ret.first << " " << ret.second << endl;

ret = chinese\_remainder\_theorem (VI(xs+3, xs+5), VI(as+3, as+5));

cout << ret.first << " " << ret.second << endl;

}

FFT\_new.cpp 16/34

#include <cassert>

#include <cstdio>

#include <cmath>

struct cpx {

cpx(){}

cpx(double aa):a(aa){}

cpx(double aa, double bb):a(aa),b(bb){}

double a;

double b;

double modsq(void) const

{

return a \* a + b \* b;

}

cpx bar(void) const

{

return cpx(a, -b);

}

};

cpx operator +(cpx a, cpx b) {

return cpx(a.a + b.a, a.b + b.b);

}

cpx operator \*(cpx a, cpx b) {

return cpx(a.a \* b.a - a.b \* b.b, a.a \* b.b + a.b \* b.a);

}

cpx operator /(cpx a, cpx b) {

cpx r = a \* b.bar();

return cpx(r.a / b.modsq(), r.b / b.modsq());

}

cpx EXP(double theta) {

return cpx(cos(theta),sin(theta));

}

const double two\_pi = 4 \* acos(0);

// in: input array

// out: output array

// step: {SET TO 1} (used internally)

// size: length of the input/output {MUST BE A POWER OF 2}

// dir: either plus or minus one (direction of the FFT)

// RESULT: out[k] = \sum\_{j=0}^{size - 1} in[j] \* exp(dir \* 2pi \* i \* j \* k / size)

void FFT(cpx \*in, cpx \*out, int step, int size, int dir) {

if(size < 1) return;

if(size == 1)

{

out[0] = in[0];

return;

}

FFT(in, out, step \* 2, size / 2, dir);

FFT(in + step, out + size / 2, step \* 2, size / 2, dir);

for(int i = 0 ; i < size / 2 ; i++)

{

cpx even = out[i];

cpx odd = out[i + size / 2];

out[i] = even + EXP(dir \* two\_pi \* i / size) \* odd;

out[i + size / 2] = even + EXP(dir \* two\_pi \* (i + size / 2) / size) \* odd;

}

}

// Usage:

// f[0...N-1] and g[0..N-1] are numbers

// Want to compute the convolution h, defined by

// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).

// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.

// Let F[0...N-1] be FFT(f), and similarly, define G and H.

// The convolution theorem says H[n] = F[n]G[n] (element-wise product).

// To compute h[] in O(N log N) time, do the following:

// 1. Compute F and G (pass dir = 1 as the argument).

// 2. Get H by element-wise multiplying F and G.

// 3. Get h by taking the inverse FFT (use dir = -1 as the argument)

// and \*dividing by N\*. DO NOT FORGET THIS SCALING FACTOR.

int main(void) {

printf("If rows come in identical pairs, then everything works.\n");

cpx a[8] = {0, 1, cpx(1,3), cpx(0,5), 1, 0, 2, 0};

cpx b[8] = {1, cpx(0,-2), cpx(0,1), 3, -1, -3, 1, -2};

cpx A[8];

cpx B[8];

FFT(a, A, 1, 8, 1);

FFT(b, B, 1, 8, 1);

for(int i = 0 ; i < 8 ; i++)

{

printf("%7.2lf%7.2lf", A[i].a, A[i].b);

}

printf("\n");

for(int i = 0 ; i < 8 ; i++)

{

cpx Ai(0,0);

for(int j = 0 ; j < 8 ; j++)

{

Ai = Ai + a[j] \* EXP(j \* i \* two\_pi / 8);

}

printf("%7.2lf%7.2lf", Ai.a, Ai.b);

}

printf("\n");

cpx AB[8];

for(int i = 0 ; i < 8 ; i++)

AB[i] = A[i] \* B[i];

cpx aconvb[8];

FFT(AB, aconvb, 1, 8, -1);

for(int i = 0 ; i < 8 ; i++)

aconvb[i] = aconvb[i] / 8;

for(int i = 0 ; i < 8 ; i++)

{

printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);

}

printf("\n");

for(int i = 0 ; i < 8 ; i++)

{

cpx aconvbi(0,0);

for(int j = 0 ; j < 8 ; j++)

{

aconvbi = aconvbi + a[j] \* b[(8 + i - j) % 8];

}

printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);

}

printf("\n");

return 0;

}

Simplex.cc 17/34

// 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] <= 0) 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;

DOUBLE \_A[m][n] = {

{ 6, -1, 0 },

{ -1, -5, 0 },

{ 1, 5, 1 },

{ -1, -5, -1 }

};

DOUBLE \_b[m] = { 10, -4, 5, -5 };

DOUBLE \_c[n] = { 1, -1, 0 };

VVD A(m);

VD b(\_b, \_b + m);

VD c(\_c, \_c + n);

for (int i = 0; i < m; i++) A[i] = VD(\_A[i], \_A[i] + n);

LPSolver solver(A, b, c);

VD x;

DOUBLE value = solver.Solve(x);

cerr << "VALUE: "<< value << endl;

cerr << "SOLUTION:";

for (size\_t i = 0; i < x.size(); i++) cerr << " " << x[i];

cerr << endl;

return 0;

}

EulerianPath.cc 20/34

struct Edge;

typedef list<Edge>::iterator iter;

struct Edge {

int next\_vertex;

iter reverse\_edge;

Edge(int next\_vertex)

:next\_vertex(next\_vertex)

{ }

};

const int max\_vertices = ;

int num\_vertices;

list<Edge> adj[max\_vertices]; // adjacency list

vector<int> path;

void find\_path(int v) {

while(adj[v].size() > 0)

{

int vn = adj[v].front().next\_vertex;

adj[vn].erase(adj[v].front().reverse\_edge);

adj[v].pop\_front();

find\_path(vn);

}

path.push\_back(v);

}

void add\_edge(int a, int b) {

adj[a].push\_front(Edge(b));

iter ita = adj[a].begin();

adj[b].push\_front(Edge(a));

iter itb = adj[b].begin();

ita->reverse\_edge = itb;

itb->reverse\_edge = ita;

}

KDTree.cc 24/34

// --------------------------------------------------------------------------

// A straightforward, but probably sub-optimal KD-tree implmentation that's

// probably good enough for most things (current it's a 2D-tree)

//

// - constructs from n points in O(n lg^2 n) time

// - handles nearest-neighbor query in O(lg n) if points are well distributed

// - worst case for nearest-neighbor may be linear in pathological case

//

// Sonny Chan, Stanford University, April 2009

// --------------------------------------------------------------------------

#include <iostream>

#include <vector>

#include <limits>

#include <cstdlib>

using namespace std;

// number type for coordinates, and its maximum value

typedef long long ntype;

const ntype sentry = numeric\_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D

struct point {

ntype x, y;

point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}

};

bool operator==(const point &a, const point &b) {

return a.x == b.x && a.y == b.y;

}

// sorts points on x-coordinate

bool on\_x(const point &a, const point &b) {

return a.x < b.x;

}

// sorts points on y-coordinate

bool on\_y(const point &a, const point &b) {

return a.y < b.y;

}

// squared distance between points

ntype pdist2(const point &a, const point &b) {

ntype dx = a.x-b.x, dy = a.y-b.y;

return dx\*dx + dy\*dy;

}

// bounding box for a set of points

struct bbox {

ntype x0, x1, y0, y1;

bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

// computes bounding box from a bunch of points

void compute(const vector<point> &v) {

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

x0 = min(x0, v[i].x); x1 = max(x1, v[i].x);

y0 = min(y0, v[i].y); y1 = max(y1, v[i].y);

}

}

// squared distance between a point and this bbox, 0 if inside

ntype distance(const point &p) {

if (p.x < x0) {

if (p.y < y0) return pdist2(point(x0, y0), p);

else if (p.y > y1) return pdist2(point(x0, y1), p);

else return pdist2(point(x0, p.y), p);

}

else if (p.x > x1) {

if (p.y < y0) return pdist2(point(x1, y0), p);

else if (p.y > y1) return pdist2(point(x1, y1), p);

else return pdist2(point(x1, p.y), p);

}

else {

if (p.y < y0) return pdist2(point(p.x, y0), p);

else if (p.y > y1) return pdist2(point(p.x, y1), p);

else return 0;

}

}

};

// stores a single node of the kd-tree, either internal or leaf

struct kdnode {

bool leaf; // true if this is a leaf node (has one point)

point pt; // the single point of this is a leaf

bbox bound; // bounding box for set of points in children

kdnode \*first, \*second; // two children of this kd-node

kdnode() : leaf(false), first(0), second(0) {}

~kdnode() { if (first) delete first; if (second) delete second; }

// intersect a point with this node (returns squared distance)

ntype intersect(const point &p) {

return bound.distance(p);

}

// recursively builds a kd-tree from a given cloud of points

void construct(vector<point> &vp)

{

// compute bounding box for points at this node

bound.compute(vp);

// if we're down to one point, then we're a leaf node

if (vp.size() == 1) {

leaf = true;

pt = vp[0];

}

else {

// split on x if the bbox is wider than high (not best heuristic...)

if (bound.x1-bound.x0 >= bound.y1-bound.y0)

sort(vp.begin(), vp.end(), on\_x);

// otherwise split on y-coordinate

else

sort(vp.begin(), vp.end(), on\_y);

// divide by taking half the array for each child

// (not best performance if many duplicates in the middle)

int half = vp.size()/2;

vector<point> vl(vp.begin(), vp.begin()+half);

vector<point> vr(vp.begin()+half, vp.end());

first = new kdnode(); first->construct(vl);

second = new kdnode(); second->construct(vr);

}

}

};

// simple kd-tree class to hold the tree and handle queries

struct kdtree {

kdnode \*root;

// constructs a kd-tree from a points (copied here, as it sorts them)

kdtree(const vector<point> &vp) {

vector<point> v(vp.begin(), vp.end());

root = new kdnode();

root->construct(v);

}

~kdtree() { delete root; }

// recursive search method returns squared distance to nearest point

ntype search(kdnode \*node, const point &p)

{

if (node->leaf) {

// commented special case tells a point not to find itself

// if (p == node->pt) return sentry;

// else

return pdist2(p, node->pt);

}

ntype bfirst = node->first->intersect(p);

ntype bsecond = node->second->intersect(p);

// choose the side with the closest bounding box to search first

// (note that the other side is also searched if needed)

if (bfirst < bsecond) {

ntype best = search(node->first, p);

if (bsecond < best)

best = min(best, search(node->second, p));

return best;

}

else {

ntype best = search(node->second, p);

if (bfirst < best)

best = min(best, search(node->first, p));

return best;

}

}

// squared distance to the nearest

ntype nearest(const point &p) {

return search(root, p);

}

};

// --------------------------------------------------------------------------

// some basic test code here

int main() {

// generate some random points for a kd-tree

vector<point> vp;

for (int i = 0; i < 100000; ++i) {

vp.push\_back(point(rand()%100000, rand()%100000));

}

kdtree tree(vp);

// query some points

for (int i = 0; i < 10; ++i) {

point q(rand()%100000, rand()%100000);

cout << "Closest squared distance to (" << q.x << ", " << q.y << ")"

<< " is " << tree.nearest(q) << endl;

}

return 0;

}

// --------------------------------------------------------------------------

SegmentTreeLazy.java 25/34

public class SegmentTreeRangeUpdate {

public long[] leaf;

public long[] update;

public int origSize;

public SegmentTreeRangeUpdate(int[] list) {

origSize = list.length;

leaf = new long[4\*list.length];

update = new long[4\*list.length];

build(1,0,list.length-1,list);

}

public void build(int curr, int begin, int end, int[] list) {

if(begin == end)

leaf[curr] = list[begin];

else {

int mid = (begin+end)/2;

build(2 \* curr, begin, mid, list);

build(2 \* curr + 1, mid+1, end, list);

leaf[curr] = leaf[2\*curr] + leaf[2\*curr+1];

}

}

public void update(int begin, int end, int val) {

update(1,0,origSize-1,begin,end,val);

}

public void update(int curr, int tBegin, int tEnd, int begin, int end, int val) {

if(tBegin >= begin && tEnd <= end)

update[curr] += val;

else {

leaf[curr] += (Math.min(end,tEnd)-Math.max(begin,tBegin)+1) \* val;

int mid = (tBegin+tEnd)/2;

if(mid >= begin && tBegin <= end)

update(2\*curr, tBegin, mid, begin, end, val);

if(tEnd >= begin && mid+1 <= end)

update(2\*curr+1, mid+1, tEnd, begin, end, val);

}

}

public long query(int begin, int end) {

return query(1,0,origSize-1,begin,end);

}

public long query(int curr, int tBegin, int tEnd, int begin, int end) {

if(tBegin >= begin && tEnd <= end) {

if(update[curr] != 0) {

leaf[curr] += (tEnd-tBegin+1) \* update[curr];

if(2\*curr < update.length){

update[2\*curr] += update[curr];

update[2\*curr+1] += update[curr];

}

update[curr] = 0;

}

return leaf[curr];

}

else {

leaf[curr] += (tEnd-tBegin+1) \* update[curr];

if(2\*curr < update.length){

update[2\*curr] += update[curr];

update[2\*curr+1] += update[curr];

}

update[curr] = 0;

int mid = (tBegin+tEnd)/2;

long ret = 0;

if(mid >= begin && tBegin <= end)

ret += query(2\*curr, tBegin, mid, begin, end);

if(tEnd >= begin && mid+1 <= end)

ret += query(2\*curr+1, mid+1, tEnd, begin, end);

return ret;

}

}

}

LogLan.java 29/34

// Code which demonstrates the use of Java's regular expression libraries.

// This is a solution for

//

// Loglan: a logical language

// http://acm.uva.es/p/v1/134.html

//

// In this problem, we are given a regular language, whose rules can be

// inferred directly from the code. For each sentence in the input, we must

// determine whether the sentence matches the regular expression or not. The

// code consists of (1) building the regular expression (which is fairly

// complex) and (2) using the regex to match sentences.

import java.util.\*;

import java.util.regex.\*;

public class LogLan {

public static String BuildRegex (){

String space = " +";

String A = "([aeiou])";

String C = "([a-z&&[^aeiou]])";

String MOD = "(g" + A + ")";

String BA = "(b" + A + ")";

String DA = "(d" + A + ")";

String LA = "(l" + A + ")";

String NAM = "([a-z]\*" + C + ")";

String PREDA = "(" + C + C + A + C + A + "|" + C + A + C + C + A + ")";

String predstring = "(" + PREDA + "(" + space + PREDA + ")\*)";

String predname = "(" + LA + space + predstring + "|" + NAM + ")";

String preds = "(" + predstring + "(" + space + A + space + predstring + ")\*)";

String predclaim = "(" + predname + space + BA + space + preds + "|" + DA + space +

preds + ")";

String verbpred = "(" + MOD + space + predstring + ")";

String statement = "(" + predname + space + verbpred + space + predname + "|" +

predname + space + verbpred + ")";

String sentence = "(" + statement + "|" + predclaim + ")";

return "^" + sentence + "$";

}

public static void main (String args[]){

String regex = BuildRegex();

Pattern pattern = Pattern.compile (regex);

Scanner s = new Scanner(System.in);

while (true) {

// In this problem, each sentence consists of multiple lines, where the last

// line is terminated by a period. The code below reads lines until

// encountering a line whose final character is a '.'. Note the use of

//

// s.length() to get length of string

// s.charAt() to extract characters from a Java string

// s.trim() to remove whitespace from the beginning and end of Java string

//

// Other useful String manipulation methods include

//

// s.compareTo(t) < 0 if s < t, lexicographically

// s.indexOf("apple") returns index of first occurrence of "apple" in s

// s.lastIndexOf("apple") returns index of last occurrence of "apple" in s

// s.replace(c,d) replaces occurrences of character c with d

// s.startsWith("apple) returns (s.indexOf("apple") == 0)

// s.toLowerCase() / s.toUpperCase() returns a new lower/uppercased string

//

// Integer.parseInt(s) converts s to an integer (32-bit)

// Long.parseLong(s) converts s to a long (64-bit)

// Double.parseDouble(s) converts s to a double

String sentence = "";

while (true){

sentence = (sentence + " " + s.nextLine()).trim();

if (sentence.equals("#")) return;

if (sentence.charAt(sentence.length()-1) == '.') break;

}

// now, we remove the period, and match the regular expression

String removed\_period = sentence.substring(0, sentence.length()-1).trim();

if (pattern.matcher (removed\_period).find()){

System.out.println ("Good");

} else {

System.out.println ("Bad!");

}

}

}

}

IO.cpp 31/34

#include <iostream>

#include <iomanip>

using namespace std;

int main() {

// Ouput a specific number of digits past the decimal point,

// in this case 5

cout.setf(ios::fixed); cout << setprecision(5);

cout << 100.0/7.0 << endl;

cout.unsetf(ios::fixed);

// Output the decimal point and trailing zeros

cout.setf(ios::showpoint);

cout << 100.0 << endl;

cout.unsetf(ios::showpoint);

// Output a '+' before positive values

cout.setf(ios::showpos);

cout << 100 << " " << -100 << endl;

cout.unsetf(ios::showpos);

// Output numerical values in hexadecimal

cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;

}

LatLong.cpp 33/34

/\*

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;

}