-------- ./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;

}

-------- ./Algorithms/Ad hoc/Bit Manipulation.h --------

#define bits(a) \_\_builtin\_popcount(a)

#define toggleBit(n, b) ((n) ^= (p2((b))))

#define LSB(n) ((n) & (-(n)))

//returns nearest power of two, choose the bigger one if both have the same difference

#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;

}

-------- ./Algorithms/Ad hoc/C++ Regex.h --------

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);

-------- ./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; }

-------- ./Algorithms/Ad hoc/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; } \

}

-------- ./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.push\_back(item);

} else {

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

\*it = item;

}

}

vi lis;

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

lis.push\_back(v[i]);

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

return lis;

}

-------- ./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;

}

-------- ./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;

}

-------- ./Algorithms/Ad hoc/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 "?";

}

-------- ./Algorithms/Ad hoc/Shunting Yard.cpp --------

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();

}

}

-------- ./Algorithms/Ad hoc/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; }

-------- ./Algorithms/Ad hoc/untitled --------

-------- ./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 low, num, parent, strongPoints;

int counter, root, rootChildren;

void dfs(Graph &g, int v) {

low[v] = num[v] = counter++;

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

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

parent[edge->to] = v;

if(v == root) rootChildren++;

dfs(g, edge->to);

if(low[edge->to] >= num[v]) strongPoints[v] = true;

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

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

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

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

}

}

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

counter = 0;

num = vi(g.V, -1), low = vi(g.V, 0), parent = vi(g.V, -1), strongPoints = vi(g.V, 0);

FOR(i, 0, g.V)

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

root = i, rootChildren = 0;

dfs(g, i);

strongPoints[root] = rootChildren > 1;

}

return strongPoints;

}

-------- ./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], edges)

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

negativeCycle = true;

return distanceTo;

}

-------- ./Algorithms/Graphs/DAGs/Shortest-Longest Path.h --------

vi shortestPath(Graph &g) {

vi order = topologicalSort(g);

vi distanceTo(g.V, INF);

FOR(i, 0, g.V) {

if(g.nodes[order[i]].inDegree == 0)

distanceTo[order[i]] = 0;

int cv = order[i];

FORC(g.edges[cv], edge)

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

}

return distanceTo;

}

-------- ./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;

}

-------- ./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;

}

-------- ./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 dfs(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;

dfs(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)

dfs(g, i, parent, state);

}

-------- ./Algorithms/Graphs/Eulerian Path.h --------

void dfs(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;

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

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

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

break;

}

}

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

}

}

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

}

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

list<int> path;

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

vi p;

FORC(path, it)

p.pb(\*it);

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

}

-------- ./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]);

}

-------- ./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)]]); }

};

-------- ./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;

}

-------- ./Algorithms/Graphs/Minimum Spanning Tree.h --------

int \*comparator;

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

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

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

UnionFindDS ds(V);

comparator = 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\* comparator;

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

comparator = &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, i));

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;

}

-------- ./Algorithms/Graphs/Networks/Dinic.cpp --------

/////UNTESTED/////

bool buildLevelGraph(Graph &g, int S, int T, vi &level) {

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

level[S] = 1;

while(!q.empty()) {

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

FOR(to, 0, g.V)

if(g.edges[cv][to].cap - g.edes[cv][to].flow > 0 || g.edges[to][cv].flow > 0 && !level[to])

q.push(to), level[to] = level[cv] + 1;

}

return level[T];

}

int constructBlockingFlow(Graph &g, int S, int T) {

int flow = 0;

stack<int> st; st.push(S);

vi visited(g.V, 0);

while(!st.empty()) {

int cv = st.top(); st.pop();

if(cv != T) {

FOR(to, 0, g.V) {

if(st.top() != T) {

if(!visited[to] && level[to] == level[cv] + 1) {

if(g.edges[cv][to].cap - g.edges[cv][to].flow > 0)

st.push(next), path[next] = cv;

else if(g.edges[to][cv].flow)

st.push(next), path[next] = -cv;

}

}

}

if(st.top() == cv)

st.pop(), visited[cv] = 1;

} else {

int F = INF, bottleneck;

for(int cur = T; cur != S; cur = abs(path[cur]))

F = min(F, path[cur] > 0 ? g.edges[path[cur]][cur].cap - g.edges[path[cur]][cur].flow : g.edges[cur][-path[cur]].flow);

for(int cur = T; cur != S; cur = abs(path[cur])) {

if(path[cur] > 0) {

g.edges[path[cur]][cur] += F;

if(g.edges[path[cur]][cur].cap - g.edges[path[cur]][cur].flow)

bottleneck = path[cur];

} else {

g.edges[path[cur]][cur] += F;

if(g.edges[cur][-path[cur]] == 0)

bottleneck = -path[cur];

}

}

while(!st.empty() && st.top() != bottleneck)

st.pop();

flow += F;

}

}

return flow;

}

int Dinic(Graph &g, int S, int T) {

int mf = 0;

vi level;

while(level.assign(g.V, 0), buildLevelGraph(g, S, T, level))

mf += constructBlockingFlow(g, S, T);

return mf;

}

-------- ./Algorithms/Graphs/Networks/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;

}

-------- ./Algorithms/Graphs/Strongly Connected Components.h --------

vi low, num, components;

int counter, SCCindex;

vector<bool> visited;

stack<int> S;

void dfs(Graph &g, int cv) {

low[cv] = num[cv] = counter++;

S.push(cv);

visited[cv] = true;

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

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

dfs(g, edge->to);

if(visited[edge->to])

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

}

if(low[cv] == num[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\*/) {

counter = 0, SCCindex = 0;

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

num = vi(g.V, -1), low = vi(g.V, 0), components = vi(g.V, 0);

S = stack<int>();

FOR(i, 0, g.V)

if(num[i] == -1)

dfs(g, i);

return components;

}

-------- ./Algorithms/Graphs/Tree Height For Each Root.cpp --------

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];

}

}

}

}

-------- ./Algorithms/Graphs/TreeHash.cpp --------

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

int treeStructureHash(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;

}

-------- ./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];

}

-------- ./Algorithms/Mathematics/Catalan Numbers.h --------

unsigned long long nthCatalan(int n) {

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

}

unsigned long long nextCatalan(int n, unsigned long long previous) {

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

}

-------- ./Algorithms/Mathematics/Cycle Finding.h --------

// 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);

}

-------- ./Algorithms/Mathematics/Euclid.h --------

// 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);

}

-------- ./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;

}

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;

}

-------- ./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);

}

-------- ./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) {

Matriz m(2, 2);

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

Matriz fib0(2, 1);

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

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

return r[1][0];

}

-------- ./Algorithms/Mathematics/Matrices.h --------

typedef vector<vector<double> > Matrix;

#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;

}

-------- ./Algorithms/Mathematics/ModPow.h --------

typedef long long ll;

ll mod(ll a, ll b) {

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

}

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

base = mod(base, modulus);

ll result = 1;

while (exp) {

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

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

exp >>= 1;

}

return result;

}

-------- ./Algorithms/Mathematics/Nth Permutation.h --------

//seq must be sorted

string nthPermutation(string seq, long long permNum) {

if(!seq.length())

return "";

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

int q = permNum / f;

long long r = permNum % f;

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

}

-------- ./Algorithms/Mathematics/Primes.h --------

#define MAX 1000000

bitset<MAX> sieve;

void buildSieve() {

sieve.set();

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

int root = sqrt(MAX);

FOR(i, 2, MAX)

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

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

sieve[j] = 0;

}

vi primesList;

void buildPrimesList() {

if(!sieve[2])

buildSieve();

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

FOR(i, 2, MAX+1)

if(sieve[i])

primesList.push\_back(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.push\_back(ii(pf, 1));

}

pf = primesList[++idx];

}

if(N!=1) factors.push\_back(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;

}

-------- ./Algorithms/Search/Binary Search.h --------

#define ANY

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

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 {

#ifdef UPPERBOUND

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

return mid;

left = mid+1;

#endif

#ifdef LOWERBOUND

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

return mid;

right = mid-1;

#endif

#ifdef ANY

return mid;

#endif

}

}

return -1;

}

-------- ./Algorithms/Sorting/Merge Sort.h --------

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

int inversions = 0;

int sorted[N];

int p1 = low, p2 = mid+1, psorted = low; //pointer to arr 1, to arr2 and to sorted arr

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];

return inversions;

}

//returns the number of inversions

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

if(low < high) {

int mid = (low + high)/2;

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

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

}

return 0;

}

-------- ./Algorithms/Sorting/Quick Sort.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);

}

-------- ./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];

}

-------- ./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;

}

-------- ./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;

}

-------- ./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];

}

-------- ./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; }

};

-------- ./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; }

};

-------- ./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;

}

};

-------- ./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);

}

-------- ./Data Structures/Geometry/Point.h --------

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 x < p.x || (x == p.x && y < p.y) || (x == p.x && y == p.y && z < p.z);

}

};

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);

}

-------- ./Data Structures/Geometry/Polygons.h --------

typedef vector<Point> Polygon;

typedef long long ll;

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.push\_back(Q[i]);

if (left \* left2 < -EPS)

P.push\_back(lineIntersectSeg(Q[i], Q[i+1], a, b));

}

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

P.push\_back(P.front());

return P;

}

-------- ./Data Structures/Geometry/Triangles.h --------

struct Triangle {

Point A, B, C;

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

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);

}

-------- ./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(2, 2);

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

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

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

Matrix vec(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)));

}

-------- ./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; }

};

-------- ./Data Structures/Lists Graph.h --------

struct Edge {

int to, weight;

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

};

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);

}

}

};

-------- ./Data Structures/Matrix Graph.h --------

struct Edge {

int weight;

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

};

struct Graph {

int V; bool undirected;

vector<vector<Edge> > edges;

Graph(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;

}

};

-------- ./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(); }

};

-------- ./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];

}

};

-------- ./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(300, n), 0), tempSA(n);

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

}

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

}

int main() {

char S[MAX\_N];

scanf("%s", S);

int n = strlen(S);

S[n] = '$';

n++;

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];

}

}

}

-------- ./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);

}

};

-------- ./Data Structures/Union Find Disjoint Sets.h --------

struct UnionFindDS {

vi tree;

UnionFindDS(int n) { FOR(i, 0, n) tree.push\_back(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 UnionFindDS {

vi tree, sizes;

int N;

UnionFindDS(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;

}

}

};

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

/\*

\*/

#include <algorithm>

#include <bitset>

#include <cmath>

#include <cstdio>

#include <cstring>

#include <deque>

#include <iomanip>

#include <iostream>

#include <queue>

#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;

mint main() {

}

-------- ./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