**Locos**

ACM ICPC 2014-2015

Notebook Part 1

Alberto Manuel Astiazarán Tobin

Alfredo Altamirano Montealvo

Salvador Pámanes Guevara

# Tabla de contenido

run.sh 4

vimrc 4

header 4

Base Conversions 5

Bit Manipulation.h 5

Dates 5

Longest Increasing Subsequence 5

Maximum Subarray 6

Range Or 6

Shunting Yard 6

Articulation Points And Bridges 7

Bellman Ford 8

Shortest-longest Path in a DAG 8

Topological Sort 8

Dijkstra 9

Edge Property Check 9

Edmonds Karp 10

Eulerian Path 10

Floyd Warshall 11

Lowest Common Ancestor 11

Maximum Bipartite Matching 12

Minimum Spanning Tree 12

Strongly Connected Components 13

Tree Hash 14

Tree Height for each Root 14

Binomial Coefficients 15

Catalan Numbers 15

Cycle Finding 15

Euclid 16

Euclid Chinese Remainder 16

FactMod 17

Fast Exponentiation 17

Fibonacci 17

Matrices 17

ModPow 19

Nth Permutation 19

Primes 20

Binary Search 21

Mergesort 21

Quicksort 22

Edit Distance 22

Longest Common Subsequence 22

String Matching 23

Subsequence Counter 23

Balanced Binary Search Tree 24

Binary Heap 25

Fenwick Tree 26

Lines 26

Point 27

Polygons 28

Triangles 30

Vectors 30

Interval Tree 31

Interval Tree SB 33

Lists Graph 35

Matrix Graph 35

Segment Tree 35

Sparse Table 37

Splay Tree 37

Suffix Array 39

Trie 40

Union Find Disjoint Sets 41

Edmonds Graph Matching 42

Roman Numerals 44

FastIO 44

Notes 44

Formulas 48

# run.sh

clear;

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

rm p

# vimrc

syntax on

nnoremap <F9> :w<Enter>:!./run.sh %<Enter>

colorscheme elflord

set mouse=a

set ai si sw=4 ts=4

set nu

set backspace=start,indent,eol

set clipboard=unnamed

set ignorecase

set smartcase

set incsearch

set scrolloff=3

highlight linenr ctermbg=darkblue

set hi=100

# header

/\*

\*/

#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;

# Base Conversions

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;

}

# 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;

}

# Dates

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; }

# Longest Increasing Subsequence

vi longestIncreasingSubsequence(vi v) {

vii best;

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

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

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

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

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;

}

# Maximum Subarray

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;

}

# Range Or

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;

}

# Shunting Yard

void output(ostream &out, string x) {

out << x << " ";

}

string readToken(istream &in) {

string t; int c;

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

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

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

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

} return t;

}

#define LEFT 0

#define RIGHT 1

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

void shunting(istream &in, ostream &out) {

string token;

stack<string> ops;

map<string, int> prec;

prec["^"] = 6;

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

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

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

assoc["^"] = RIGHT;

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

if(isOp(token)) {

while(!ops.empty() && isOp(ops.top())

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

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

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

ops.push(token);

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

ops.push(token);

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

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

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

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

ops.pop();

} else // numbers vars

output(out, token);

}

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

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

}

}

# Articulation Points And Bridges

//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;

}

# Bellman Ford

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;

}

# Shortest-longest Path in a DAG

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;

}

# Topological Sort

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;

}

# Dijkstra

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;

}

# Edge Property Check

#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);

}

# Edmonds Karp

int augment(MatrixGraph &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(MatrixGraph &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;

}

# Eulerian Path

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;

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);

if(g.edges[initial].size()%2 == 0)

path.pop\_front();

vi p;

FORC(path, it)

p.pb(\*it);

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

}

# Floyd Warshall

#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]);

}

# Lowest Common Ancestor

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)]]); }

};

# Maximum Bipartite Matching

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))

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

return 0;

}

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

//g must be unweighted directed bipartite graph

//match[r] = l, where r belongs to R and l belongs to L

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;

}

# Minimum Spanning Tree

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;} };

//Returns a list of edges (node, indexOfEdge)

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;

}

# Strongly Connected Components

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;

}

# Tree Hash

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;

}

# Tree Height for each Root

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];

}

}

}

}

# Binomial Coefficients

//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];

}

# Catalan Numbers

int fact(int n) {

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

}

int nthCatalan(int n) {

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

}

int nextCatalan(int n, int previous) {

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

}

# Cycle Finding

int f(int i) { return (7\*i+5)%12; }

// 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);

}

# Euclid

// 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);

}

# Euclid Chinese Remainder

// returns x = a\_i (mod n\_i)

// n's must be pairwise coprimes

int chinese\_remainder(int \*n, int \*a, int len) {

int p, i, prod = 1, sum = 0;

for (i = 0; i < len; i++) prod \*= n[i];

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

p = prod / n[i];

sum += a[i] \* mod\_inverse(p, n[i]) \* p;

}

return sum % prod;

}

# FactMod

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;

}

# Fast Exponentiation

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);

}

# Fibonacci

int fibn(int n) { //max 91

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

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

}

int 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];

}

# Matrices

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;

}

# ModPow

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;

}

# Nth Permutation

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);

}

# Primes

#define SIZE 1000000

bitset<SIZE> sieve;

void buildSieve() {

sieve.set();

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

int root = sqrt(SIZE);

FOR(i, 2, root+1)

if (sieve[i])

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(int N) {

vii factors;

int 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, int 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(ll N) {

vii pf = primeFactorization(N);

vi div;

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

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

return div;

}

bool isPrime(int 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;

}

# Binary Search

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;

}

# Mergesort

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;

}

# Quicksort

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);

}

# Edit Distance

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];

}

# Longest Common Subsequence

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;

}

# String Matching

vi buildTable(string& pattern) {

vi table(pattern.length()+1);

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;

}

# Subsequence Counter

// 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];

}

# Balanced Binary Search Tree

#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; }

};

# Binary Heap

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; }

};

# Fenwick Tree

//1 based indexing

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) { if(R<r)swap(r,R); if(C<c)swap(c, C);return query(R, C) - query(r-1, C) - query(R, c-1) + query(r-1, c-1); }

void add(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;

}

};

# Lines

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);

}

# Point

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);

}

# Polygons

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;

}

# Triangles

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 area(const Triangle &T) {

double ab = dist(T.A, T.B);

double bc = dist(T.B, T.C);

double ca = dist(T.C, T.A);

return area(ab, bc, ca);

}

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(const Triangle &T) {

Point A = T.A, B = T.B, C = T.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);

}

# Vectors

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)));

}

# Interval Tree

#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), left(NULL), right(NULL) {}

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

};

# Interval Tree SB

#ifndef LCHILD

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

#define \_LCHILD(n) ((n)->parent->\*left == (n))

#endif

#define FIX(n) (n ? (maxsec = max(maxsec, n->maxsec)) : 0)

#define ICONTAINS(i, v) ((v) >= (i).first && (v) <= (i).second)

#define IOVERLAPS(a, b) ((a).first <= (b).second && (a).second >= (b).first)

template< typename T, typename Compare = less<pair<T, T> > >

class IntervalSplayTree {

typedef pair<T, T> K;

Compare compare;

struct Node {

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

K key;

T maxsec;

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

void fix() {

T mx = maxsec;

maxsec = key.second;

FIX(left); FIX(right);

if(maxsec != mx && parent) parent->fix();}

};

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);

parent->fix();

}

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;

if(node->parent) pred->parent->fix();

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;

if(node->parent) node->parent->fix();

delete node;

}

}

void rotate(Node \*parent, bool rleft) {

Node \*Node::\*left = &Node::left, \*Node::\*right = &Node::right;

if(!rleft) swap(left, right);

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;

parent->fix();

}

void splay(Node \*node) {

while(root != node) {

if(node->parent->parent) {

bool lcnode = LCHILD(node), lcparent = LCHILD(node->parent);

rotate(lcnode == lcparent ? node->parent->parent : node->parent, !lcnode);

rotate(node->parent, !lcparent);

} else rotate(node->parent, !LCHILD(node));

}

}

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

public:

IntervalSplayTree() : root(0) {}

~IntervalSplayTree() { 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; }

void search(Node \*n, K p, set<K> &result) {

if (!n) return;

if (p.first > n->maxsec) return; // p > maxsec

if (n->left) search(n->left, p, result);

if (IOVERLAPS(n->key, p))result.insert(n->key);

if (p.second< n->key.first) return;

if (n->right) search(n->right, p, result);

}

set<K> intersections(K i) {

set<K> res; search(root, i, res);

return res;

}

bool overlaps(K i) {

return !intersections(i).empty();

}

};

# Lists Graph

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

}

}

};

# Matrix Graph

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, Edge edge = Edge(1)) {

edges[from][to] = edge;

if(undirected) edges[to][from] = edge;

}

};

# Segment Tree

vi values;

struct Node {

int sum;

Node() { init(); }

Node(int pos, int value) { init(); update(0, 0, 0, 0, value); }

void init() {

sum = 0;

#ifdef LAZY

lazy = 0; hasUpdates = false;

#endif

}

void update(int L, int R, int from, int to, int value) {

if(from > R || to < L) return;

sum += (min(to, R) - max(from, L) + 1)\*value;

}

int ans() { return sum; }

Node operator+(Node &rNode) { return Node(0, sum + rNode.sum); }

#ifdef LAZY

int lazy; bool hasUpdates;

void storeUpdate(int value) { lazy += value; hasUpdates = true; }

void applyUpdates(int L, int R) { if(hasUpdates) update(L, R, L, R, lazy); lazy = 0; hasUpdates = false; }

int getUpdates() { return lazy; }

#endif

};

struct SegmentTree {

vector<Node> tree;

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

#ifdef LAZY

if(L != R && tree[index].hasUpdates) {

update(index\*2, L, (L+R)/2, 0, values.size(), tree[index].getUpdates());

update(index\*2+1, (L+R)/2+1, R, 0, values.size(), tree[index].getUpdates());

}

tree[index].applyUpdates(L, R);

#endif

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

Node left, right;

bool queryL = false, queryR = false;

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

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

return !queryL ? right : (!queryR ? left : left + right);

}

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

#ifdef LAZY

void update(int index, int L, int R, int from, int to, int value) {

if (from > R || to < L) return;

if(L >= from && R <= to) { tree[index].storeUpdate(value); return; }

tree[index].update(L, R, from, to, value);

update(index\*2, L, (L+R)/2, from, to, value);

update(index\*2+1, (L+R)/2+1, R, from, to, value);

}

void update(int from, int to, int value) { update(1, 0, values.size()-1, from, to, value); }

#endif

void pointUpdate(int index, int L, int R, int pos, int value) {

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

if(L == R) { tree[index] = Node(pos, value); return; }

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

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

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

}

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

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

if(L == R) { tree[index] = Node(L, values[L]); return; }

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

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

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

}

SegmentTree(vi A) {

tree.clear();

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

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

initialize(1, 0, A.size()-1, 0, A.size()-1);

}

};

# Sparse Table

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];

}

};

# Splay Tree

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

#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 rotate(Node \*parent, bool rleft) {

Node \*Node::\*left = &Node::left, \*Node::\*right = &Node::right;

if(!rleft) swap(left, right);

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 splay(Node \*node) {

while(root != node) {

if(node->parent->parent) {

bool lcnode = LCHILD(node), lcparent = LCHILD(node->parent);

rotate(lcnode == lcparent ? node->parent->parent : node->parent, !lcnode);

rotate(node->parent, !lcparent);

} else rotate(node->parent, !LCHILD(node));

}

}

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; }

};

# Suffix Array

#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];

}

}

}

\*/

# Trie

#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);

}

};

# Union Find Disjoint Sets

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;

}

}

};

# Edmonds Graph Matching

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;

}

}

}

# Roman Numerals

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 "?";

}

# FastIO

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; } \

}

# Notes

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);

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

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; }

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

vector<int> v(3, 5); //init vector to {5, 5, 5}

int arr[] = {2, 3, 4};

vector<int> v(arr, arr+3); //init vector to array

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

int\* it = lower\_bound(arr, arr+N, searchValue/\*optional: comparator (it must be the same comparator used to sort)\*/)

if(it == arr+N) cout << "not found" << endl;

else cout << "found " << \*it << " at index " << it-arr << endl;

lower\_bound: finds first that does not compare less than val.

upper\_bound: finds first that compares greater than val.

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

int arr[] = {1, 2, 3}

reverse(arr, arr+N); //reverses the array, arr = {3, 2, 1}

sort(arr+N, arr) //reverse sort

partial\_sort(arr, arr+k, arr+N) //partially sorts the array time: klog(N)

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

struct Point

{

double x, y;

string id;

};

Point origin = {0, 0, "origin"};

Point points[3] = {{3.4, 2.1, "myPoint1"},

{2.4, 7.2, "myPoint2"},

{4.1, 8.1, "myPoint3"}};

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

#include <algorithm>

int arr[] = {0, 1, 2, 3, 4};

next\_permutation(arr, arr+5); //0, 1, 2, 4, 3

next\_permutation(arr, arr+5); //0, 1, 3, 2, 4

prev\_permutation(...)

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

#include <map>

#include <set>

//check if it contains an item

myMap.count(item);

mySet.count(item);

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

//When sorting small structs, for example:

struct Team

{

int goldMedals;

int silverMedals;

int bronzeMedals;

};

//sort by gold, then silver then bronze

//instead of defining a comparison function, another way is to:

typedef pair<int, pair<int, int> > Team;

Team teams[10];

teams[0] = make\_pair(4, make\_pair(2, 6));

...

sort(teams, teams + 10);

//drawback: all variables will be sorted ascending or descending

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

#include <iomanip>

cout << fixed << setprecision(3) << 23.2341 << endl; //23.234 //formats forever until changed

cout.setwidth(8); //only for the next cout

cout << 2355 << endl; //"2355" -> " 2355"

cout.fill("-"); //forever until changed again

cout << 2355 << endl; //"2355" -> "------2355"

cout.setwidth(10);

cout << left << 2355 << endl; //"2355" -> "2355------"

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

scanf:

%d -> base10 int | %d+

%o -> base8 int | %d+

%x -> base16 int | %d+

%a -> base10 or base16 double | ex. 123, 34.24, 5464.324e+3, 53423E+2, 0x242.435, base16 if preceded by 0x

%c -> char or array of chars | ex. scanf("%c", &mychar) -> 'a', scanf("%4c", mycharptr) -> "asdf" (\0 not included)

%s -> string

matching: scanf("abc%d", &myint) with input: "ab34 ascz24 abc345" would store 345 in myint, use %% to match %

%\*d means match an int but dont store it in a parameter

%3d means match an integer but read only the 3 first characters

%lld stores in a long long int %d matches int, more specifiers are:

%le long double

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

class comparator

{

bool operator()(int a, int b)

{

return a < b;

}

}

priority\_queue<int, vector<int>, comparator> pq;

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

Bellman-Ford for solving system of inequalities of the type x\_i - x\_j <= c

create a node for every x

create a source node

create a zero weight edge from s to every other node

for every inequality x\_i - x\_j <= c, make an edge from i to j of weight c

run bellman ford starting at s

the value for x\_i is d\_i

if there was a negative weight cycle, the system is inconsistent

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

max\_element(arr, arr+N); //returns a pointer to the element

min\_element(arr, arr+N);

//arrays must be sorted, can be used with set and map

merge() //same as set union but allows duplicates

set\_union()//A + B

set\_intersection()//A intersection B

set\_difference()// A - B

set\_symmetric\_difference() // A^B

parameters: (begin1, end1, begin2, end2, begin\_result); //returns a pointer to the end of the result, and the result is stored [begin\_result to end\_result)

accumulate(arr, arr+N, (long long)0); //add all elements by default, function can be specified, 0 is the initial value

double product = accumulate(all(v), double(1), multiplies<double>());

//plus, minus, divides, modulus, negate, equal\_to, custom functions implemented same way as priority queue comparator

inner\_product(all(v1), v2.begin(), 0); //scalar product [a, b, c].[d, e, f] = a\*d + b\*e + c\*f

for\_each(vec.begin(), vec.end(), func); //calls func(i) for every element in [begin, end)

nth\_element (vec.begin(), vec.begin()+n, vec.end(), myfunction);

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

#include <ctype.h>

isalpha(char c), isupper(char c) ,islower(char c), isdigit(char c), ispunct(char c), toupper(char c), tolower(char c)

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

In a bipartite graph, the size of the maximum independent set (or dominating set) = V-MCBM

In a bipartite graph, the size of the min vertex cover = MCBM

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

char str[] = "abc. sdfksm sgfda afdex.. NJK- ,, . hb564567....";

char \* token = strtok(str, ". ");

while (token != NULL)

{

printf ("%s\n",token);

token = strtok (NULL, ". ");

}

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

There are n^(n-2) spanning trees in a complete graph with n vertices

A dearangement is a permutation of a set where all elements are in a different position than their original position

der(n) = (n-1)\*(der(n-1)+der(n-2)), der(0) = 1, der(1) = 0

a finite sequence of natural numbers can be a degree sequence of a graph iff the sum is even and sum from i=1 to k of d\_i < k\*(k-1) for 1<=k<=n

E - V - 2 = F where F is the number of faces in a planar graph

The number of pieces in which a circle is divided if n points on its circumference are joined by chords with no three internally concurrent:

g(n) = nCat4 + nCat2 + 1

A = i+b/2-1 where A is the area of a polygon, i is the number of integer points on the polygon and b is the number of integer points on the boundary

the number of spanning trees in complete a bipartite graph K(n, m) is m^(n-1) \* n^(m-1)

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

//splitting by spaces

istringstream iss(line);

vector<string> tokens;

copy(istream\_iterator<string>(iss), istream\_iterator<string>(), back\_inserter<vector<string> >(tokens));

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

loop adjacent elements in a grid

//8 de alrededor

FOR(k, 0, 9)

if(k != 4) {

ii next(i+k%2-1, j+k/3-1);

}

//arriba/abajo/der/izq: for(int k = 1; k < 9; k+=2)

FOR(k, 0, 4) {

int k2 = k\*2+1;

ii next(i+k2%2-1, j+k2/3-1);

}

~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~

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

# Formulas

**Catalan Numbers**

**Heron’s formula**

, **where**

**Law of cosine**

**Law of sine**

**Newton Raphson**

**Series: Arithmetic**

**Series: Geometric**

If |r| < 1

**Simpson’s Rule**

**Stirling’s approximation**

**Sum of Powers**