# 

**Locos**

ACM ICPC 2013-2014

Formulario

Alberto Manuel Astiazarán Tobin

Alfredo Altamirano Montealvo

Diego Emilio Gutiérrez Yépiz

Index

Data Structures 2

Adj. Lists Graph 2

Adj. Matrix Graph 2

Balanced Binary Search Tree 2

Binary Heap 2

Fenwick Tree / 2D-FT 3

Segment Tree 3

Sparse Table 3

Suffix Array 3

Trie 4

Union-Find Disjoint Sets 4

Graphs 4

Articulation Points and Bridges 4

BFS 4

Bipartite Graph Check 5

Bipartite: Maximum Bipartite Matching 5

DFS 5

DAG: Topological Sort 5

DAG: Single Source Shortest/Longest Path 5

Edges Property Check 5

Eulerian Path 5

MST Kruskal 6

MST Prim 6

Network: Max Flow (Edmonds-Karp) 6

Shortest Path: SS: Dijkstra 6

Shortest Path: SS: Bellman-Ford 6

Shortest Path: AP: Floyd-Warshall 6

Strongly Connected Components 6

Tree: Lowest Common Ancestor 7

Sorting 7

Mergesort 7

Quicksort 7

Searching 7

Binary Search 7

Mathematics 7

Binomial Coefficients 7

Cycle Finding (Floyd) 7

Euclid 7

Fast Exponentiation 8

Fibonacci 8

Matrices 8

Modpow 9

Prime Numbers 9

Strings 9

Edit Distance 9

Longest Common Subsequence 9

String Matching (Knuth-Morris-Pratt) 9

Subsequence Counter 9

Geometry 9

Lines 9

Point 10

Polygon 10

Triangles 11

Vectors 11

Miscellaneous 11

Base conversion 11

Bit manipulation 11

Dates 11

Expression Parsing/Evaluating 12

Hungarian Algorithm 12

Longest Increasing Subsequence 14

Maximum Subarray (Kadane) 14

Range OR 14

Regex 15

String-Number Conversion 15

Formulas 15

Catalan Numbers 15

Heron’s formula 15

Law of cosine 15

Law of sine 15

Newton Raphson 15

Series: Arithmetic 15

Series: Geometric 15

Simpson’s Rule 15

Stirling’s approximation 15

Sum of Powers 15

Stanford’s Notebook 15

Dinic 15

MinCostMaxFlow 15

PushRelabel 16

MinCostMatching 17

MaxBipartiteMatching 17

MinCut 17

Geometry 18

Geom3D.java 20

Delaunay Triangulation 20

Simplex 21

KDTree 22

Edmonds Graph Matching 23

Roman Numerals 24

Hamiltonian Path 24

Factorial Modulo 25

# Data Structures

## Adj. Lists Graph

struct Edge {

int to, weight;

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

};

struct Graph {

int V;

vector<vector<Edge> > edges;

bool undirected;

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

void connect(int from, Edge edge) {

edges[from].push\_back(edge);

if(undirected) {

int aux = edge.to;

edge.to = from;

edges[aux].push\_back(edge);

}

}

};

## Adj. Matrix Graph

struct Edge {

int weight;

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

};

struct Graph {

int V;

vector<vector<Edge> > edges;

bool undirected;

Graph(int v, bool undirected = true) : V(v), undirected(undirected) {

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

}

void connect(int from, int to, Edge edge = Edge()) {

edges[from][to] = edge;

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

}

};

## 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 / 2D-FT

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;

}

};

## Segment Tree

int operation(int a, int b) { return a+b; }

//Space: 2\*2^(floor(log\_2(N))+1)(\*2 with lazy propagation)

struct SegmentTree {

private:

int N;

vi tree, lazy;

void update(int treeIndex, int L, int R, int element, int value) {

if(L == R) {

tree[treeIndex] = value;

return;

}

if(element <= (L+R)/2)

update(treeIndex\*2, L, (L+R)/2, element, value);

else if(element >= (L+R)/2 + 1)

update(treeIndex\*2 + 1, (L+R)/2 + 1, R, element, value);

tree[treeIndex] = operation(tree[treeIndex\*2], tree[treeIndex\*2 + 1]);

}

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

if(L >= from && R <= to)

return tree[treeIndex];

int 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;

else if(!queryR) return left;

return operation(left, right);

}

void rangeUpdate(int treeIndex, int L, int R, int from, int to, int value) {

if(lazy[treeIndex] != 0) {

tree[treeIndex] += lazy[treeIndex];

if(L != R) {

lazy[treeIndex\*2] += lazy[treeIndex];

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

}

lazy[treeIndex] = 0;

}

if(L == R) {

tree[treeIndex] += value;

return;

}

if(L >= from && R <= to) {

tree[treeIndex] += value\*(to-from+1);

if(L != R) {

lazy[treeIndex\*2] += value;

lazy[treeIndex\*2+1] += value;

}

return;

}

if(from <= (L+R)/2)

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

if(to >= (L+R)/2 + 1)

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

tree[treeIndex] = operation(tree[treeIndex\*2], tree[treeIndex\*2 + 1]);

}

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

if(L == R) {

tree[treeIndex] = values[L];

return;

}

if(from <= (L+R)/2)

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

if(to >= (L+R)/2 + 1)

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

tree[treeIndex] = operation(tree[treeIndex\*2], tree[treeIndex\*2 + 1]);

}

public:

SegmentTree(int values[], int N) : N(N) {

tree.clear();

tree.assign(2\*(1<<(int(log(N)/log(2))+1)), 0);

lazy.assign(2\*(1<<(int(log(N)/log(2))+1)), 0);

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

}

void update(int i, int k) { update(1, 0, N-1, i, k); }

int query(int from, int to) { return query(1, 0, N-1, from, to); }

};

## 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];

}

};

## Suffix Array

#define MAX\_N 100000

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];

}

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]];

}

## 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.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;

}

}

};

# Graphs

## Articulation Points and Bridges

void dfs(Graph &g, int currentVertex, vi &low, vi &num, vi &parent, vi &strongPoints, int &counter, int root, int &rootChildren) {

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

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

int neighbor = g.edges[currentVertex][i].to;

if(num[neighbor] == 0) {

parent[neighbor] = currentVertex;

if(neighbor == root)

rootChildren++;

dfs(g, neighbor, low, num, parent, strongPoints, counter, root, rootChildren);

if(low[neighbor] >= num[currentVertex])

strongPoints[currentVertex] = true;

if(low[neighbor] > num[currentVertex])

g.edges[currentVertex][i].strong = true;

low[currentVertex] = min(low[currentVertex], low[neighbor]);

} else if(neighbor != parent[currentVertex])

low[currentVertex] = min(low[currentVertex], num[neighbor]);

}

}

//Must be an undirected graph

vi articulationPointsAndBridges(Graph &g) {

int counter = 0, root, rootChildren;

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

FOR(i, 0, g.V)

if(num[i] == 0) {

root = i, rootChildren = 0;

dfs(g, i, low, num, parent, strongPoints, counter, root, rootChildren);

strongPoints[root] = rootChildren > 1;

}

return strongPoints;

}

## BFS

void BFS(Graph &g, int startingVertex) {

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

visited[startingVertex] = 1;

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

while(!q.empty()) {

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

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

int neighbor = g.edges[currentVertex][i].to;

if(!visited[neighbor]) {

visited[neighbor] = 1;

q.push(neighbor);

}

}

}

}

## Bipartite Graph Check

bool isBipartite(Graph &g) {

queue<int> q;

q.push(0);

vi color(g.V, -1);

color[0] = 0;

while(!q.empty()) {

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

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

int neighbor = g.edges[currentVertex][i].to;

if(color[neighbor] == -1) {

color[neighbor] = !color[currentVertex];

q.push(neighbor);

} else if(color[neighbor] == color[currentVertex])

return false;

}

}

return true;

}

## Bipartite: Maximum Bipartite Matching

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

if(visited[cv]) return 0;

visited[cv] = 1;

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

int neighbor = g.edges[cv][i].to;

if(match[neighbor] == -1 || augment(g, match[neighbor], match, visited)) {

match[neighbor] = 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;

}

## DFS

void DFS(Graph &g, int currentVertex, int visited[]) {

visited[currentVertex] = 1;

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

int neighbor = g.edges[currentVertex][i].to;

if(!visited[neighbor]) DFS(g, neighbor, visited);

}

}

## DAG: Topological Sort

vi topologicalSort(Graph &g) {

vi order;

int inDegree[g.V];

FOR(i, 0, g.V) {

inDegree[i] = g.nodes[i].inDegree;

if(inDegree[i] == 0)

order.push\_back(i);

}

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

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

if(--inDegree[g.edges[order[i]][j].to] == 0)

order.push\_back(g.edges[order[i]][j].to);

return order;

}

void dfs(Graph &g, int currentVertex, vi &order, bool visited[])

{

visited[currentVertex] = true;

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

{

int neighbor = g.edges[currentVertex][i].to;

if(!visited[neighbor])

dfs(g, neighbor, order, visited);

}

order.push\_back(currentVertex);

}

//Recursive version

vi topologicalSort2(Graph &g)

{

vi order;

bool visited[g.V];

memset(visited, 0, sizeof visited);

FOR(i, 0, g.V)

if(visited[i] == 0)

dfs(g, i, order, visited);

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

return order;

}

## DAG: Single Source Shortest/Longest Path

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 currentVertex = order[i];

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

int neighbor = g.edges[currentVertex][j].to;

distanceTo[neighbor] = min(distanceTo[neighbor], g.edges[currentVertex][j].weight + distanceTo[currentVertex]);

}

}

return distanceTo;

}

## Edges 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 dfs(Graph &g, int currentVertex, vi &parent, vi &state) {

state[currentVertex] = EXPLORED;

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

int neighbor = g.edges[currentVertex][i].to;

if(state[neighbor] == UNVISITED) {

g.edges[currentVertex][i].type = TREE;

parent[neighbor] = currentVertex;

dfs(g, neighbor, parent, state);

} else if(state[neighbor] == EXPLORED) {

if(neighbor != parent[currentVertex]) //if they are equal it is a bidirectional edge

g.edges[currentVertex][i].type = BACK;

}

else if(state[neighbor] == VISITED)

g.edges[currentVertex][i].type = FORWARD;

}

state[currentVertex] = 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);

}

## Eulerian Path

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

bool last = true;

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

int neighbor = g.edges[currentVertex][i].to;

if(!g.edges[currentVertex][i].visited) {

last = false;

g.edges[currentVertex][i].visited = 1;

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

int neighborOfNeighbor = g.edges[neighbor][j].to;

if(neighborOfNeighbor == currentVertex && !g.edges[neighbor][j].visited) {

g.edges[neighbor][j].visited = 1;

break;

}

}

dfs(g, path, path.insert(it, currentVertex), neighbor);

}

}

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

}

//only for undirected graphs

vi getEulerianPath(Graph &g, int initial)

{

list<int> path;

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

vi p;

for(list<int>::iterator it=path.begin(); it!=path.end(); it++)

p.push\_back(\*it);

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

return p;

}

## MST Kruskal

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.push\_back(order[i]);

}

return minTree;

}

## MST Prim

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 currentVertex = 0;

while(visitedNodes != g.V) {

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

if(!visited[g.edges[currentVertex][i].to])

pq.push(ii(currentVertex, i));

ii nextEdge;

do {

nextEdge = pq.top();

pq.pop();

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

tree.push\_back(nextEdge);

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

visitedNodes++;

visited[currentVertex] = 1;

}

return tree;

}

## Network: Max Flow (Edmonds-Karp)

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

if(currentVertex == source)

return minEdge;

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

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

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

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

}

return flow;

}

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

if(currentVertex == sink) break;

FOR(i, 0, g.V) {

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

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

}

}

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

mf += flow;

}

return mf;

}

## Shortest Path: SS: Dijkstra

vi dijkstra(Graph &g, int source) {

vi distanceTo(g.V, INF);

distanceTo[source] = 0;

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

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

while(!pq.empty()) {

int currentVertex = pq.top().second;

int d = pq.top().first;

pq.pop();

if(d > distanceTo[currentVertex]) continue;

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

int neighbor = g.edges[currentVertex][i].to;

if(distanceTo[neighbor] > distanceTo[currentVertex] + g.edges[currentVertex][i].weight) {

distanceTo[neighbor] = distanceTo[currentVertex] + g.edges[currentVertex][i].weight;

pq.push(ii(distanceTo[neighbor], neighbor));

}

}

}

return distanceTo;

}

## Shortest Path: SS: 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)

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

distanceTo[g.edges[j][k].to] = min(distanceTo[g.edges[j][k].to], distanceTo[j] + g.edges[j][k].weight);

//to detect negative weight cycles:

FOR(i, 0, g.V)

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

if(distanceTo[g.edges[i][j].to] > distanceTo[i] + g.edges[i][j].weight)

negativeCycle = true;

return distanceTo;

}

## Shortest Path: AP: 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]);

}

## Strongly Connected Components

void dfs(Graph &g, int currentVertex, vi &low, vi &num, int &counter, vector<bool> visited, stack<int> &S, UnionFindDS &sets) {

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

S.push(currentVertex);

visited[currentVertex] = true;

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

int neighbor = g.edges[currentVertex][i].to;

if(num[neighbor] == 0)

dfs(g, neighbor, low, num, counter, visited, S, sets);

if(visited[neighbor])

low[currentVertex] = min(low[currentVertex], low[neighbor]);

}

if(low[currentVertex] == num[currentVertex]) {

int v, root = S.top(); S.pop(); visited[root] = false;

v = root;

while(v != currentVertex) {

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

sets.connect(root, v);

}

}

}

//the graph must be directed

UnionFindDS stronglyConnectedComponents(Graph &g)

{

UnionFindDS sets(g.V);

int counter = 0, SCCindex = 0;

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

vi num(g.V, 0), low(g.V, 0);

stack<int> S;

FOR(i, 0, g.V)

if(num[i] == 0)

dfs(g, i, low, num, counter, visited, S, sets);

return sets;

}

## Tree: 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.push\_back(cv), height.push\_back(h);

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

if(index[g.edges[cv][i].to] == -1) {

dfs(g, g.edges[cv][i].to, height.back() + g.edges[cv][i].weight, index);

order.push\_back(cv), height.push\_back(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)]]); }

};

# Sorting

## Mergesort

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;

}

## Quicksort

//array must be shuffled before sorting

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);

}

# Searching

## Binary Search

#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;

}

# Mathematics

## Binomial Coefficients

//max 25

long long fact(int n) {

long long res = 1;

FOR(i, 2, n+1) res\*=i;

return res;

}

int nCr(long long n, long long r) {

long long res = 1;

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

res \*= n + i;

return res/fact(r);

}

#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];

}

## Cycle Finding (Floyd)

/ // 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

// 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);

}

## 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

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];

}

## Matrices

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;

}

## Modpow

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;

}

## Prime Numbers

vii primeFactorization(long long N, vi primes) {

vii factors;

long long idx = 0, pf = primes[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 = primes[++idx];

}

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

return factors;

}

#define MAX 32000

bitset<MAX> sieveOfAtkin() {

bitset<MAX> sieve;

int max = sieve.size()-1;

sieve[2] = sieve[3] = 1;

int root = ceil(sqrt(max));

for (int x = 1; x <= root; x++)

for (int y = 1; y <= root; y++) {

int xsq = x\*x, ysq = y\*y;

int n = 4\*xsq + ysq;

if (n <= max && (n%12 == 1 || n%12 == 5)) sieve[n] = !sieve[n];

n -= xsq;

if (n <= max && n%12 == 7) sieve[n] = !sieve[n];

n -= 2\*ysq;

if (x > y && n <= max && n%12 == 11)

sieve[n] = !sieve[n];

}

for (int r = 5; r <= root; r+=2)

if (sieve[r])

for (int i = r\*r; i<max; i+=r\*r)

sieve[i] = 0;

return sieve;

}

vi getPrimes() {

vi primes;

bitset<MAX> sieve = sieveOfAtkin();

FOR(i, 2, MAX+1)

if(sieve[i]) primes.push\_back(i);

return primes;

}

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;

}

# Strings

## 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 (Knuth-Morris-Pratt)

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.push\_back(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];

}

# Geometry

## 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(a, b);

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

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);

}

## Polygon

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;

}

## Triangles

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);

}

## 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(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)));

}

# Miscellaneous

## Base conversion

string toBaseN(long long num, int N) {

string converted;

for(long long div=num; div; div /= N) {

int value = div % N;

char c = value > 9 ? value + 'A' - 10 : value + '0';

converted = c + converted;

}

return converted;

}

long long toBase10(string num, int b) {

long long res = 0, k = 1;

for(int i=num.length()-1; i>=0; i--) {

char c = toUpper(num[i]);

int value = c > '9' ? c - 'A' + 10 : c - '0';

res += (value)\*k;

k\*=b;

}

return res;

}

## Bit manipulation

#define bits(a) \_\_builtin\_popcount(a) //quick bit count on unsigned int

#define isOn(n, b) (((n) & (p2(b))) != 0) //checks if bit b is on

#define setBit(n, b) ((n) |= p2(b)) //sets bit b on n

#define clearBit(n, b) ((n) &= ~(p2((b)))) //clears bit b on n

#define toggleBit(n, b) ((n) ^= (p2((b)))) //toggles bit b on n

#define LSB(n) ((n) & (-(n))) //returns the least significant bit that is on

#define setAll(n, b) ((n) = p2(b) - 1) //sets n to b ones

#define modulo(x, N) ((x) & (N-1)) //returns x % N, where N is a power of 2

#define isPowerOfTwo(x) (((x) & ((x)-1)) == 0)

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

#define nearestPowerOfTwo(x) ((int)pow(2.0, (int)((log((double)x) / log(2.0)) + 0.5)))

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

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

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

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

void printBin(int N) {

stack<int> st;

while (N) st.push(N & 1), N >>= 1;

while (!st.empty()) printf("%d", st.top()), st.pop();

printf("\n");

}

## 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; }

## Expression Parsing/Evaluating

typedef int (\*Operation)(int, int);

struct Operator {

Operation operation;

int priority;

Operator() {}

Operator(Operation op, int pr) { operation = op, priority = pr;}

};

int add(int a, int b) { return a+b; }

int subtract(int a, int b) { return a-b; }

int multiply(int a, int b) { return a\*b; }

int divide(int a, int b) { return a/b; }

map<char, Operator> \_operator;

void initOperators() {

\_operator['+'] = Operator(add, 2);

\_operator['-'] = Operator(subtract, 2);

\_operator['\*'] = Operator(multiply, 3);

\_operator['/'] = Operator(divide, 3);

}

bool isOperand(char d) { return d >= '0' && d <= '9'; }

bool isOperator(char o) { return \_operator.find(o) != \_operator.end(); }

template<typename T> T pop(stack<T> &st) { T top = st.top(); st.pop(); return top; }

template<typename T> T pop(queue<T> &q) { T front = q.front(); q.pop(); return front; }

const int PREFIX = 0, POSTFIX = 1;

string convert(string infix, int conversionTo) {

if(\_operator.size() == 0)

initOperators();

string result;

stack<char> st;

if(conversionTo == PREFIX)

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

for(int i=0; i<infix.length(); i++) {

char token = infix[i];

if(conversionTo == PREFIX) {

if(token == '(') token = ')';

else if(token == ')') token = '(';

}

if(isOperand(token)) result += token;

else if(token == '(') st.push(token);

else if(token == ')')

while((token = pop(st)) != '(')

result += token;

else if(isOperator(token))

{

while(!st.empty() && isOperator(st.top()) &&

(\_operator[st.top()].priority > \_operator[token].priority ||

conversionTo == POSTFIX && \_operator[st.top()].priority == \_operator[token].priority))

result += pop(st);

st.push(token);

}

}

while(!st.empty())

result += pop(st);

if(conversionTo == PREFIX)

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

return result;

}

int evaluate(string postfix) {

stack<int> st;

for(int i=0; i<postfix.length(); i++) {

char token = postfix[i];

if(isOperand(token))

st.push(token-'0');

else if(isOperator(token))

{

int b = pop(st), a = pop(st);

st.push(\_operator[token].operation(a, b));

}

}

return st.top();

}

## Hungarian Algorithm

template <class T>

class Matrix {

public:

Matrix();

Matrix(int rows, int columns);

Matrix(const Matrix<T> &other);

Matrix<T> & operator= (const Matrix<T> &other);

~Matrix();

void resize(int rows, int columns);

void clear(void);

T& operator () (int x, int y);

inline int minsize(void) { return ((m\_rows < m\_columns) ? m\_rows : m\_columns); }

inline int columns(void) { return m\_columns; }

inline int rows(void) { return m\_rows; }

private:

T \*\*m\_matrix;

int m\_rows;

int m\_columns;

};

template <class T>

Matrix<T>::Matrix() {

m\_rows = 0;

m\_columns = 0;

m\_matrix = NULL;

}

template <class T>

Matrix<T>::Matrix(const Matrix<T> &other) {

if(other.m\_matrix != NULL ) {

m\_matrix = NULL;

resize(other.m\_rows, other.m\_columns);

for(int i = 0 ; i < m\_rows ; i++ )

for(int j = 0 ; j < m\_columns ; j++ )

m\_matrix[i][j] = other.m\_matrix[i][j];

} else {

m\_matrix = NULL;

m\_rows = 0;

m\_columns = 0;

}

}

template <class T>

Matrix<T>::Matrix(int rows, int columns) {

m\_matrix = NULL;

resize(rows, columns);

}

template <class T>

Matrix<T> &

Matrix<T>::operator=(const Matrix<T> &other) {

if(other.m\_matrix != NULL ) {

resize(other.m\_rows, other.m\_columns);

for(int i = 0 ; i < m\_rows ; i++ )

for(int j = 0 ; j < m\_columns ; j++ )

m\_matrix[i][j] = other.m\_matrix[i][j];

} else {

for(int i = 0 ; i < m\_columns ; i++ )

delete [] m\_matrix[i];

delete [] m\_matrix;

m\_matrix = NULL;

m\_rows = 0;

m\_columns = 0;

}

return \*this;

}

template <class T>

Matrix<T>::~Matrix() {

if(m\_matrix != NULL ) {

for(int i = 0 ; i < m\_rows ; i++ )

delete [] m\_matrix[i];

delete [] m\_matrix;

}

m\_matrix = NULL;

}

template <class T>

void Matrix<T>::resize(int rows, int columns) {

if(m\_matrix == NULL ) {

m\_matrix = new T\*[rows];

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

m\_matrix[i] = new T[columns];

m\_rows = rows;

m\_columns = columns;

clear();

} else {

T \*\*new\_matrix;

new\_matrix = new T\*[rows];

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

new\_matrix[i] = new T[columns];

for(int j = 0 ; j < columns ; j++ )

new\_matrix[i][j] = 0;

}

int minrows = std::min<int>(rows, m\_rows);

int mincols = std::min<int>(columns, m\_columns);

for(int x = 0 ; x < minrows ; x++ )

for(int y = 0 ; y < mincols ; y++ )

new\_matrix[x][y] = m\_matrix[x][y];

if(m\_matrix != NULL ) {

for(int i = 0 ; i < m\_rows ; i++ )

delete [] m\_matrix[i];

delete [] m\_matrix;

}

m\_matrix = new\_matrix;

}

m\_rows = rows;

m\_columns = columns;

}

template <class T>

void Matrix<T>::clear() {

for(int i = 0 ; i < m\_rows ; i++ )

for(int j = 0 ; j < m\_columns ; j++ )

m\_matrix[i][j] = 0;

}

template <class T>

inline T& Matrix<T>::operator ()(int x, int y) {

return m\_matrix[x][y];

}

#include <list>

class Munkres {

public:

void solve(Matrix<double> &m);

private:

static const int NORMAL = 0;

static const int STAR = 1;

static const int PRIME = 2;

inline bool find\_uncovered\_in\_matrix(double,int&,int&);

inline bool pair\_in\_list(const std::pair<int,int> &, const std::list<std::pair<int,int> > &);

int step1(void);

int step2(void);

int step3(void);

int step4(void);

int step5(void);

int step6(void);

Matrix<int> mask\_matrix;

Matrix<double> matrix;

bool \*row\_mask;

bool \*col\_mask;

int saverow, savecol;

};

bool Munkres::find\_uncovered\_in\_matrix(double item, int &row, int &col) {

for(row = 0 ; row < matrix.rows() ; row++ )

if(!row\_mask[row] )

for(col = 0 ; col < matrix.columns() ; col++ )

if(!col\_mask[col] )

if(matrix(row,col) == item )

return true;

return false;

}

bool Munkres::pair\_in\_list(const std::pair<int,int> &needle, const std::list<std::pair<int,int> > &haystack) {

for(std::list<std::pair<int,int> >::const\_iterator i = haystack.begin() ; i != haystack.end() ; i++ )

if(needle == \*i )

return true;

return false;

}

int Munkres::step1(void) {

for(int row = 0 ; row < matrix.rows() ; row++ )

for(int col = 0 ; col < matrix.columns() ; col++ )

if(matrix(row,col) == 0 ) {

bool isstarred = false;

for(int nrow = 0 ; nrow < matrix.rows() ; nrow++ )

if(mask\_matrix(nrow,col) == STAR ) {

isstarred = true;

break;

}

if(!isstarred ) {

for(int ncol = 0 ; ncol < matrix.columns() ; ncol++ )

if(mask\_matrix(row,ncol) == STAR ) {

isstarred = true;

break;

}

}

if(!isstarred )

mask\_matrix(row,col) = STAR;

}

return 2;

}

int Munkres::step2(void) {

int rows = matrix.rows();

int cols = matrix.columns();

int covercount = 0;

for(int row = 0 ; row < rows ; row++ )

for(int col = 0 ; col < cols ; col++ )

if(mask\_matrix(row,col) == STAR ) {

col\_mask[col] = true;

covercount++;

}

int k = matrix.minsize();

if(covercount >= k ) {

return 0;

}

return 3;

}

int Munkres::step3(void) {

if(find\_uncovered\_in\_matrix(0, saverow, savecol) )

mask\_matrix(saverow,savecol) = PRIME;

else

return 5;

for(int ncol = 0 ; ncol < matrix.columns() ; ncol++ )

if(mask\_matrix(saverow,ncol) == STAR ) {

row\_mask[saverow] = true;

col\_mask[ncol] = false;

return 3;

}

return 4;

}

int Munkres::step4(void) {

int rows = matrix.rows();

int cols = matrix.columns();

std::list<std::pair<int,int> > seq;

std::pair<int,int> z0(saverow, savecol);

std::pair<int,int> z1(-1,-1);

std::pair<int,int> z2n(-1,-1);

seq.insert(seq.end(), z0);

int row, col = savecol;

bool madepair;

do {

madepair = false;

for(row = 0 ; row < rows ; row++ )

if(mask\_matrix(row,col) == STAR ) {

z1.first = row;

z1.second = col;

if(pair\_in\_list(z1, seq) )

continue;

madepair = true;

seq.insert(seq.end(), z1);

break;

}

if(!madepair )

break;

madepair = false;

for(col = 0 ; col < cols ; col++ )

if(mask\_matrix(row,col) == PRIME ) {

z2n.first = row;

z2n.second = col;

if(pair\_in\_list(z2n, seq) )

continue;

madepair = true;

seq.insert(seq.end(), z2n);

break;

}

} while(madepair );

for(std::list<std::pair<int,int> >::iterator i = seq.begin() ;

i != seq.end() ;

i++ ) {

if(mask\_matrix(i->first,i->second) == STAR )

mask\_matrix(i->first,i->second) = NORMAL;

if(mask\_matrix(i->first,i->second) == PRIME )

mask\_matrix(i->first,i->second) = STAR;

}

for(int row = 0 ; row < mask\_matrix.rows() ; row++ )

for(int col = 0 ; col < mask\_matrix.columns() ; col++ )

if(mask\_matrix(row,col) == PRIME )

mask\_matrix(row,col) = NORMAL;

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

row\_mask[i] = false;

}

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

col\_mask[i] = false;

}

return 2;

}

int Munkres::step5(void) {

int rows = matrix.rows();

int cols = matrix.columns();

double h = 0;

for(int row = 0 ; row < rows ; row++ ) {

if(!row\_mask[row] ) {

for(int col = 0 ; col < cols ; col++ ) {

if(!col\_mask[col] ) {

if((h > matrix(row,col) && matrix(row,col) != 0) || h == 0 ) {

h = matrix(row,col);

}

}

}

}

}

for(int row = 0 ; row < rows ; row++ )

if(row\_mask[row] )

for(int col = 0 ; col < cols ; col++ )

matrix(row,col) += h;

for(int col = 0 ; col < cols ; col++ )

if(!col\_mask[col] )

for(int row = 0 ; row < rows ; row++ )

matrix(row,col) -= h;

return 3;

}

void Munkres::solve(Matrix<double> &m) {

double highValue = 0;

for(int row = 0 ; row < m.rows() ; row++ ) {

for(int col = 0 ; col < m.columns() ; col++ ) {

if(m(row,col) != INFINITY && m(row,col) > highValue )

highValue = m(row,col);

}

}

highValue++;

for(int row = 0 ; row < m.rows() ; row++ )

for(int col = 0 ; col < m.columns() ; col++ )

if(m(row,col) == INFINITY )

m(row,col) = highValue;

bool notdone = true;

int step = 1;

this->matrix = m;

mask\_matrix.resize(matrix.rows(), matrix.columns());

row\_mask = new bool[matrix.rows()];

col\_mask = new bool[matrix.columns()];

for(int i = 0 ; i < matrix.rows() ; i++ ) {

row\_mask[i] = false;

}

for(int i = 0 ; i < matrix.columns() ; i++ ) {

col\_mask[i] = false;

}

while(notdone ) {

switch ( step ) {

case 0:

notdone = false;

break;

case 1:

step = step1();

break;

case 2:

step = step2();

break;

case 3:

step = step3();

break;

case 4:

step = step4();

break;

case 5:

step = step5();

break;

}

}

for(int row = 0 ; row < matrix.rows() ; row++ )

for(int col = 0 ; col < matrix.columns() ; col++ )

if(mask\_matrix(row,col) == STAR )

matrix(row,col) = 0;

else

matrix(row,col) = -1;

m = matrix;

delete [] row\_mask;

delete [] col\_mask;

}

## Longest Increasing Subsequence

#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;

}

## Maximum Subarray (Kadane)

int maximumSubarray(int numbers, int N) {

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

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

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

else maxEndingHere += numbers[i];

maxSoFar = max(maxEndingHere, maxSoFar);

}

return maxSoFar;

}

## Range OR

typedef unsigned long long ull;

ull rangeOR(ull A, ull B) {

ull value = 0;

for(ull i=1ull << 63; i; i >>= 1) {

value <<= 1;

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

}

return value;

}

## Regex

## String-Number Conversion

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; }

# 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

# Stanford’s Notebook

## Dinic

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

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

// Running time:

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

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

// OUTPUT:

// - maximum flow value

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

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

struct Edge {

int from, to, cap, flow, index;

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

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

};

struct Dinic {

int N;

vector<vector<Edge> > G;

vector<Edge \*> dad;

vector<int> Q;

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

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

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

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

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

}

long long BlockingFlow(int s, int t) {

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

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

int head = 0, tail = 0;

Q[tail++] = s;

while (head < tail) {

int x = Q[head++];

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

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

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

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

Q[tail++] = e.to;

}

}

}

if (!dad[t]) return 0;

long long totflow = 0;

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

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

int amt = INF;

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

if (!e) { amt = 0; break; }

amt = min(amt, e->cap - e->flow);

}

if (amt == 0) continue;

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

e->flow += amt;

G[e->to][e->index].flow -= amt;

}

totflow += amt;

}

return totflow;

}

long long GetMaxFlow(int s, int t) {

long long totflow = 0;

while (long long flow = BlockingFlow(s, t))

totflow += flow;

return totflow;

}

};

## MinCostMaxFlow

// Implementation of min cost max flow algorithm using adjacency

// matrix (Edmonds and Karp 1972). This implementation keeps track of

// forward and reverse edges separately (so you can set cap[i][j] !=

// cap[j][i]). For a regular max flow, set all edge costs to 0.

// Running time, O(|V|^2) cost per augmentation

// max flow: O(|V|^3) augmentations

// min cost max flow: O(|V|^4 \* MAX\_EDGE\_COST) augmentations

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

// OUTPUT:

// - (maximum flow value, minimum cost value)

// - To obtain the actual flow, look at positive values only.

typedef vector<int> VI;

typedef vector<VI> VVI;

typedef long long L;

typedef vector<L> VL;

typedef vector<VL> VVL;

typedef pair<int, int> PII;

typedef vector<PII> VPII;

const L INF = numeric\_limits<L>::max() / 4;

struct MinCostMaxFlow {

int N;

VVL cap, flow, cost;

VI found;

VL dist, pi, width;

VPII dad;

MinCostMaxFlow(int N) :

N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),

found(N), dist(N), pi(N), width(N), dad(N) {}

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

this->cap[from][to] = cap;

this->cost[from][to] = cost;

}

void Relax(int s, int k, L cap, L cost, int dir) {

L val = dist[s] + pi[s] - pi[k] + cost;

if (cap && val < dist[k]) {

dist[k] = val;

dad[k] = make\_pair(s, dir);

width[k] = min(cap, width[s]);

}

}

L Dijkstra(int s, int t) {

fill(found.begin(), found.end(), false);

fill(dist.begin(), dist.end(), INF);

fill(width.begin(), width.end(), 0);

dist[s] = 0;

width[s] = INF;

while (s != -1) {

int best = -1;

found[s] = true;

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

if (found[k]) continue;

Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);

Relax(s, k, flow[k][s], -cost[k][s], -1);

if (best == -1 || dist[k] < dist[best]) best = k;

}

s = best;

}

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

pi[k] = min(pi[k] + dist[k], INF);

return width[t];

}

pair<L, L> GetMaxFlow(int s, int t) {

L totflow = 0, totcost = 0;

while (L amt = Dijkstra(s, t)) {

totflow += amt;

for (int x = t; x != s; x = dad[x].first) {

if (dad[x].second == 1) {

flow[dad[x].first][x] += amt;

totcost += amt \* cost[dad[x].first][x];

} else {

flow[x][dad[x].first] -= amt;

totcost -= amt \* cost[x][dad[x].first];

}

}

}

return make\_pair(totflow, totcost);

}

};

## PushRelabel

// Adjacency list implementation of FIFO push relabel maximum flow

// with the gap relabeling heuristic. This implementation is

// significantly faster than straight Ford-Fulkerson. It solves

// random problems with 10000 vertices and 1000000 edges in a few

// seconds, though it is possible to construct test cases that

// achieve the worst-case.

// Running time:

// O(|V|^3)

// INPUT:

// - graph, constructed using AddEdge()

// - source

// - sink

// OUTPUT:

// - maximum flow value

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

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

typedef long long LL;

struct Edge {

int from, to, cap, flow, index;

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

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

};

struct PushRelabel {

int N;

vector<vector<Edge> > G;

vector<LL> excess;

vector<int> dist, active, count;

queue<int> Q;

PushRelabel(int N) : N(N), G(N), excess(N), dist(N), active(N), count(2\*N) {}

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

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

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

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

}

void Enqueue(int v) {

if (!active[v] && excess[v] > 0) { active[v] = true; Q.push(v); }

}

void Push(Edge &e) {

int amt = int(min(excess[e.from], LL(e.cap - e.flow)));

if (dist[e.from] <= dist[e.to] || amt == 0) return;

e.flow += amt;

G[e.to][e.index].flow -= amt;

excess[e.to] += amt;

excess[e.from] -= amt;

Enqueue(e.to);

}

void Gap(int k) {

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

if (dist[v] < k) continue;

count[dist[v]]--;

dist[v] = max(dist[v], N+1);

count[dist[v]]++;

Enqueue(v);

}

}

void Relabel(int v) {

count[dist[v]]--;

dist[v] = 2\*N;

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

if (G[v][i].cap - G[v][i].flow > 0)

dist[v] = min(dist[v], dist[G[v][i].to] + 1);

count[dist[v]]++;

Enqueue(v);

}

void Discharge(int v) {

for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);

if (excess[v] > 0) {

if (count[dist[v]] == 1)

Gap(dist[v]);

else

Relabel(v);

}

}

LL GetMaxFlow(int s, int t) {

count[0] = N-1;

count[N] = 1;

dist[s] = N;

active[s] = active[t] = true;

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

excess[s] += G[s][i].cap;

Push(G[s][i]);

}

while (!Q.empty()) {

int v = Q.front();

Q.pop();

active[v] = false;

Discharge(v);

}

LL totflow = 0;

for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;

return totflow;

}

};

## MinCostMatching

// Min cost bipartite matching via shortest augmenting paths

// This is an O(n^3) implementation of a shortest augmenting path

// algorithm for finding min cost perfect matchings in dense

// graphs. In practice, it solves 1000x1000 problems in around 1

// second.

// cost[i][j] = cost for pairing left node i with right node j

// Lmate[i] = index of right node that left node i pairs with

// Rmate[j] = index of left node that right node j pairs with

// The values in cost[i][j] may be positive or negative. To perform

// maximization, simply negate the cost[][] matrix.

typedef vector<double> VD;

typedef vector<VD> VVD;

typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {

int n = int(cost.size());

VD u(n);

VD v(n);

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

u[i] = cost[i][0];

for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);

}

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

v[j] = cost[0][j] - u[0];

for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);

}

Lmate = VI(n, -1);

Rmate = VI(n, -1);

int mated = 0;

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

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

if (Rmate[j] != -1) continue;

if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {

Lmate[i] = j;

Rmate[j] = i;

mated++;

break;

}

}

}

VD dist(n);

VI dad(n);

VI seen(n);

while (mated < n) {

int s = 0;

while (Lmate[s] != -1) s++;

fill(dad.begin(), dad.end(), -1);

fill(seen.begin(), seen.end(), 0);

for (int k = 0; k < n; k++)

dist[k] = cost[s][k] - u[s] - v[k];

int j = 0;

while (true) {

j = -1;

for (int k = 0; k < n; k++) {

if (seen[k]) continue;

if (j == -1 || dist[k] < dist[j]) j = k;

}

seen[j] = 1;

if (Rmate[j] == -1) break;

const int i = Rmate[j];

for (int k = 0; k < n; k++) {

if (seen[k]) continue;

const double new\_dist = dist[j] + cost[i][k] - u[i] - v[k];

if (dist[k] > new\_dist) {

dist[k] = new\_dist;

dad[k] = j;

}

}

}

for (int k = 0; k < n; k++) {

if (k == j || !seen[k]) continue;

const int i = Rmate[k];

v[k] += dist[k] - dist[j];

u[i] -= dist[k] - dist[j];

}

u[s] += dist[j];

while (dad[j] >= 0) {

const int d = dad[j];

Rmate[j] = Rmate[d];

Lmate[Rmate[j]] = j;

j = d;

}

Rmate[j] = s;

Lmate[s] = j;

mated++;

}

double value = 0;

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

value += cost[i][Lmate[i]];

return value;

}

## MaxBipartiteMatching

// This code performs maximum bipartite matching.

// Running time: O(|E| |V|) -- often much faster in practice

// INPUT: w[i][j] = edge between row node i and column node j

// OUTPUT: mr[i] = assignment for row node i, -1 if unassigned

// mc[j] = assignment for column node j, -1 if unassigned

// function returns number of matches made

typedef vector<int> VI;

typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {

for (int j = 0; j < w[i].size(); j++) {

if (w[i][j] && !seen[j]) {

seen[j] = true;

if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {

mr[i] = j;

mc[j] = i;

return true;

}

}

}

return false;

}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {

mr = VI(w.size(), -1);

mc = VI(w[0].size(), -1);

int ct = 0;

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

VI seen(w[0].size());

if (FindMatch(i, w, mr, mc, seen)) ct++;

}

return ct;

}

## MinCut

// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.

// Running time:

// O(|V|^3)

// INPUT:

// - graph, constructed using AddEdge()

// OUTPUT:

// - (min cut value, nodes in half of min cut)

typedef vector<int> VI;

typedef vector<VI> VVI;

pair<int, VI> GetMinCut(VVI &weights) {

int N = weights.size();

VI used(N), cut, best\_cut;

int best\_weight = -1;

for (int phase = N-1; phase >= 0; phase--) {

VI w = weights[0];

VI added = used;

int prev, last = 0;

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

prev = last;

last = -1;

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

if (!added[j] && (last == -1 || w[j] > w[last])) last = j;

if (i == phase-1) {

for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];

for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];

used[last] = true;

cut.push\_back(last);

if (best\_weight == -1 || w[last] < best\_weight) {

best\_cut = cut;

best\_weight = w[last];

}

} else {

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

w[j] += weights[last][j];

added[last] = true;

}

}

}

return make\_pair(best\_weight, best\_cut);

}

## Geometry

double INF = 1e100;

double EPS = 1e-12;

struct PT {

double x, y;

PT() {}

PT(double x, double y) : x(x), y(y) {}

PT(const PT &p) : x(p.x), y(p.y) {}

PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }

PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }

PT operator \* (double c) const { return PT(x\*c, y\*c ); }

PT operator / (double c) const { return PT(x/c, y/c ); }

};

double dot(PT p, PT q) { return p.x\*q.x+p.y\*q.y; }

double dist2(PT p, PT q) { return dot(p-q,p-q); }

double cross(PT p, PT q) { return p.x\*q.y-p.y\*q.x; }

ostream &operator<<(ostream &os, const PT &p) {

os << "(" << p.x << "," << p.y << ")"; }

// rotate a point CCW or CW around the origin

PT RotateCCW90(PT p) { return PT(-p.y,p.x); }

PT RotateCW90(PT p) { return PT(p.y,-p.x); }

PT RotateCCW(PT p, double t) {

return PT(p.x\*cos(t)-p.y\*sin(t), p.x\*sin(t)+p.y\*cos(t)); }

// project point c onto line through a and b

// assuming a != b

PT ProjectPointLine(PT a, PT b, PT c) {

return a + (b-a)\*dot(c-a, b-a)/dot(b-a, b-a); }

// project point c onto line segment through a and b

PT ProjectPointSegment(PT a, PT b, PT c) {

double r = dot(b-a,b-a);

if (fabs(r) < EPS) return a;

r = dot(c-a, b-a)/r;

if (r < 0) return a;

if (r > 1) return b;

return a + (b-a)\*r;

}

// compute distance from c to segment between a and b

double DistancePointSegment(PT a, PT b, PT c) {

return sqrt(dist2(c, ProjectPointSegment(a, b, c))); }

// compute distance between point (x,y,z) and plane ax+by+cz=d

double DistancePointPlane(double x, double y, double z,

double a, double b, double c, double d) {

return fabs(a\*x+b\*y+c\*z-d)/sqrt(a\*a+b\*b+c\*c); }

// determine if lines from a to b and c to d are parallel or collinear

bool LinesParallel(PT a, PT b, PT c, PT d) {

return fabs(cross(b-a, c-d)) < EPS; }

bool LinesCollinear(PT a, PT b, PT c, PT d) {

return LinesParallel(a, b, c, d)

&& fabs(cross(a-b, a-c)) < EPS

&& fabs(cross(c-d, c-a)) < EPS;

}

// determine if line segment from a to b intersects with

// line segment from c to d

bool SegmentsIntersect(PT a, PT b, PT c, PT d) {

if (LinesCollinear(a, b, c, d)) {

if (dist2(a, c) < EPS || dist2(a, d) < EPS ||

dist2(b, c) < EPS || dist2(b, d) < EPS) return true;

if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)

return false;

return true;

}

if (cross(d-a, b-a) \* cross(c-a, b-a) > 0) return false;

if (cross(a-c, d-c) \* cross(b-c, d-c) > 0) return false;

return true;

}

// compute intersection of line passing through a and b

// with line passing through c and d, assuming that unique

// intersection exists; for segment intersection, check if

// segments intersect first

PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {

b=b-a; d=c-d; c=c-a;

assert(dot(b, b) > EPS && dot(d, d) > EPS);

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

}

// compute center of circle given three points

PT ComputeCircleCenter(PT a, PT b, PT c) {

b=(a+b)/2;

c=(a+c)/2;

return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));

}

// determine if point is in a possibly non-convex polygon (by William

// Randolph Franklin); returns 1 for strictly interior points, 0 for

// strictly exterior points, and 0 or 1 for the remaining points.

// Note that it is possible to convert this into an \*exact\* test using

// integer arithmetic by taking care of the division appropriately

// (making sure to deal with signs properly) and then by writing exact

// tests for checking point on polygon boundary

bool PointInPolygon(const vector<PT> &p, PT q) {

bool c = 0;

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

int j = (i+1)%p.size();

if ((p[i].y <= q.y && q.y < p[j].y ||

p[j].y <= q.y && q.y < p[i].y) &&

q.x < p[i].x + (p[j].x - p[i].x) \* (q.y - p[i].y) / (p[j].y - p[i].y))

c = !c;

}

return c;

}

// determine if point is on the boundary of a polygon

bool PointOnPolygon(const vector<PT> &p, PT q) {

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

if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)

return true;

return false;

}

// compute intersection of line through points a and b with

// circle centered at c with radius r > 0

vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {

vector<PT> ret;

b = b-a;

a = a-c;

double A = dot(b, b);

double B = dot(a, b);

double C = dot(a, a) - r\*r;

double D = B\*B - A\*C;

if (D < -EPS) return ret;

ret.push\_back(c+a+b\*(-B+sqrt(D+EPS))/A);

if (D > EPS)

ret.push\_back(c+a+b\*(-B-sqrt(D))/A);

return ret;

}

// compute intersection of circle centered at a with radius r

// with circle centered at b with radius R

vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {

vector<PT> ret;

double d = sqrt(dist2(a, b));

if (d > r+R || d+min(r, R) < max(r, R)) return ret;

double x = (d\*d-R\*R+r\*r)/(2\*d);

double y = sqrt(r\*r-x\*x);

PT v = (b-a)/d;

ret.push\_back(a+v\*x + RotateCCW90(v)\*y);

if (y > 0)

ret.push\_back(a+v\*x - RotateCCW90(v)\*y);

return ret;

}

// This code computes the area or centroid of a (possibly nonconvex)

// polygon, assuming that the coordinates are listed in a clockwise or

// counterclockwise fashion. Note that the centroid is often known as

// the "center of gravity" or "center of mass".

double ComputeSignedArea(const vector<PT> &p) {

double area = 0;

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

int j = (i+1) % p.size();

area += p[i].x\*p[j].y - p[j].x\*p[i].y;

}

return area / 2.0;

}

double ComputeArea(const vector<PT> &p) {

return fabs(ComputeSignedArea(p)); }

PT ComputeCentroid(const vector<PT> &p) {

PT c(0,0);

double scale = 6.0 \* ComputeSignedArea(p);

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

int j = (i+1) % p.size();

c = c + (p[i]+p[j])\*(p[i].x\*p[j].y - p[j].x\*p[i].y);

}

return c / scale;

}

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

bool IsSimple(const vector<PT> &p) {

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

for (int k = i+1; k < p.size(); k++) {

int j = (i+1) % p.size();

int l = (k+1) % p.size();

if (i == l || j == k) continue;

if (SegmentsIntersect(p[i], p[j], p[k], p[l]))

return false;

}

}

return true;

}

int main() {

// expected: (-5,2)

cerr << RotateCCW90(PT(2,5)) << endl;

// expected: (5,-2)

cerr << RotateCW90(PT(2,5)) << endl;

// expected: (-5,2)

cerr << RotateCCW(PT(2,5),M\_PI/2) << endl;

// expected: (5,2)

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

// expected: (5,2) (7.5,3) (2.5,1)

cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "

<< ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "

<< ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

// expected: 6.78903

cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

// expected: 1 0 1

cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "

<< LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "

<< LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

// expected: 0 0 1

cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "

<< LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "

<< LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

// expected: 1 1 1 0

cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "

<< SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "

<< SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "

<< SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

// expected: (1,2)

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

// expected: (1,1)

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

vector<PT> v;

v.push\_back(PT(0,0));

v.push\_back(PT(5,0));

v.push\_back(PT(5,5));

v.push\_back(PT(0,5));

// expected: 1 1 1 0 0

cerr << PointInPolygon(v, PT(2,2)) << " "

<< PointInPolygon(v, PT(2,0)) << " "

<< PointInPolygon(v, PT(0,2)) << " "

<< PointInPolygon(v, PT(5,2)) << " "

<< PointInPolygon(v, PT(2,5)) << endl;

// expected: 0 1 1 1 1

cerr << PointOnPolygon(v, PT(2,2)) << " "

<< PointOnPolygon(v, PT(2,0)) << " "

<< PointOnPolygon(v, PT(0,2)) << " "

<< PointOnPolygon(v, PT(5,2)) << " "

<< PointOnPolygon(v, PT(2,5)) << endl;

// expected: (1,6)

// (5,4) (4,5)

// blank line

// (4,5) (5,4)

// blank line

// (4,5) (5,4)

vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);

for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);

for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);

for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);

for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);

for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);

for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

// area should be 5.0

// centroid should be (1.1666666, 1.166666)

PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };

vector<PT> p(pa, pa+4);

PT c = ComputeCentroid(p);

cerr << "Area: " << ComputeArea(p) << endl;

cerr << "Centroid: " << c << endl;

}

JavaGeometry.java 9/27

// In this example, we read an input file containing three lines, each

// containing an even number of doubles, separated by commas. The first two

// lines represent the coordinates of two polygons, given in counterclockwise

// (or clockwise) order, which we will call "A" and "B". The last line

// contains a list of points, p[1], p[2], ...

// Our goal is to determine:

// (1) whether B - A is a single closed shape (as opposed to multiple shapes)

// (2) the area of B - A

// (3) whether each p[i] is in the interior of B - A

// INPUT:

// 0 0 10 0 0 10

// 0 0 10 10 10 0

// 8 6

// 5 1

// OUTPUT:

// The area is singular.

// The area is 25.0

// Point belongs to the area.

// Point does not belong to the area.

import java.util.\*;

import java.awt.geom.\*;

import java.io.\*;

public class JavaGeometry {

// make an array of doubles from a string

static double[] readPoints(String s) {

String[] arr = s.trim().split("\\s++");

double[] ret = new double[arr.length];

for (int i = 0; i < arr.length; i++) ret[i] = Double.parseDouble(arr[i]);

return ret;

}

// make an Area object from the coordinates of a polygon

static Area makeArea(double[] pts) {

Path2D.Double p = new Path2D.Double();

p.moveTo(pts[0], pts[1]);

for (int i = 2; i < pts.length; i += 2) p.lineTo(pts[i], pts[i+1]);

p.closePath();

return new Area(p);

}

// compute area of polygon

static double computePolygonArea(ArrayList<Point2D.Double> points) {

Point2D.Double[] pts = points.toArray(new Point2D.Double[points.size()]);

double area = 0;

for (int i = 0; i < pts.length; i++){

int j = (i+1) % pts.length;

area += pts[i].x \* pts[j].y - pts[j].x \* pts[i].y;

}

return Math.abs(area)/2;

}

// compute the area of an Area object containing several disjoint polygons

static double computeArea(Area area) {

double totArea = 0;

PathIterator iter = area.getPathIterator(null);

ArrayList<Point2D.Double> points = new ArrayList<Point2D.Double>();

while (!iter.isDone()) {

double[] buffer = new double[6];

switch (iter.currentSegment(buffer)) {

case PathIterator.SEG\_MOVETO:

case PathIterator.SEG\_LINETO:

points.add(new Point2D.Double(buffer[0], buffer[1]));

break;

case PathIterator.SEG\_CLOSE:

totArea += computePolygonArea(points);

points.clear();

break;

}

iter.next();

}

return totArea;

}

// notice that the main() throws an Exception -- necessary to

// avoid wrapping the Scanner object for file reading in a

// try { ... } catch block.

public static void main(String args[]) throws Exception {

Scanner scanner = new Scanner(new File("input.txt"));

// also,

// Scanner scanner = new Scanner (System.in);

double[] pointsA = readPoints(scanner.nextLine());

double[] pointsB = readPoints(scanner.nextLine());

Area areaA = makeArea(pointsA);

Area areaB = makeArea(pointsB);

areaB.subtract(areaA);

// also,

// areaB.exclusiveOr (areaA);

// areaB.add (areaA);

// areaB.intersect (areaA);

// (1) determine whether B - A is a single closed shape (as

// opposed to multiple shapes)

boolean isSingle = areaB.isSingular();

// also,

// areaB.isEmpty();

if (isSingle)

System.out.println("The area is singular.");

else

System.out.println("The area is not singular.");

// (2) compute the area of B - A

System.out.println("The area is " + computeArea(areaB) + ".");

// (3) determine whether each p[i] is in the interior of B - A

while (scanner.hasNextDouble()) {

double x = scanner.nextDouble();

assert(scanner.hasNextDouble());

double y = scanner.nextDouble();

if (areaB.contains(x,y)) {

System.out.println ("Point belongs to the area.");

} else {

System.out.println ("Point does not belong to the area.");

}

}

// Finally, some useful things we didn't use in this example:

// Ellipse2D.Double ellipse = new Ellipse2D.Double (double x, double y,

// double w, double h);

// creates an ellipse inscribed in box with bottom-left corner (x,y)

// and upper-right corner (x+y,w+h)

//

// Rectangle2D.Double rect = new Rectangle2D.Double (double x, double y,

// double w, double h);

// creates a box with bottom-left corner (x,y) and upper-right

// corner (x+y,w+h)

// Each of these can be embedded in an Area object (e.g., new Area (rect)).

}

}

## Geom3D.java

public class Geom3D {

// distance from point (x, y, z) to plane aX + bY + cZ + d = 0

public static double ptPlaneDist(double x, double y, double z,

double a, double b, double c, double d) {

return Math.abs(a\*x + b\*y + c\*z + d) / Math.sqrt(a\*a + b\*b + c\*c); }

// distance between parallel planes aX + bY + cZ + d1 = 0 and

// aX + bY + cZ + d2 = 0

public static double planePlaneDist(double a, double b, double c,

double d1, double d2) {

return Math.abs(d1 - d2) / Math.sqrt(a\*a + b\*b + c\*c); }

// distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)

// (or ray, or segment; in the case of the ray, the endpoint is the

// first point)

public static final int LINE = 0;

public static final int SEGMENT = 1;

public static final int RAY = 2;

public static double ptLineDistSq(double x1, double y1, double z1,

double x2, double y2, double z2, double px, double py, double pz,

int type) {

double pd2 = (x1-x2)\*(x1-x2) + (y1-y2)\*(y1-y2) + (z1-z2)\*(z1-z2);

double x, y, z;

if (pd2 == 0) {

x = x1;

y = y1;

z = z1;

} else {

double u = ((px-x1)\*(x2-x1) + (py-y1)\*(y2-y1) + (pz-z1)\*(z2-z1)) / pd2;

x = x1 + u \* (x2 - x1);

y = y1 + u \* (y2 - y1);

z = z1 + u \* (z2 - z1);

if (type != LINE && u < 0) {

x = x1;

y = y1;

z = z1;

}

if (type == SEGMENT && u > 1.0) {

x = x2;

y = y2;

z = z2;

}

}

return (x-px)\*(x-px) + (y-py)\*(y-py) + (z-pz)\*(z-pz);

}

public static double ptLineDist(double x1, double y1, double z1,

double x2, double y2, double z2, double px, double py, double pz,

int type) {

return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz, type)); }

}

## Delaunay Triangulation

// Slow but simple Delaunay triangulation. Does not handle

// degenerate cases (from O'Rourke, Computational Geometry in C)

// Running time: O(n^4)

// INPUT: x[] = x-coordinates

// y[] = y-coordinates

// OUTPUT: triples = a vector containing m triples of indices

// corresponding to triangle vertices

typedef double T;

struct triple {

int i, j, k;

triple() {}

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

};

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {

int n = x.size();

vector<T> z(n);

vector<triple> ret;

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

z[i] = x[i] \* x[i] + y[i] \* y[i];

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

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

for (int k = i+1; k < n; k++) {

if (j == k) continue;

double xn = (y[j]-y[i])\*(z[k]-z[i]) - (y[k]-y[i])\*(z[j]-z[i]);

double yn = (x[k]-x[i])\*(z[j]-z[i]) - (x[j]-x[i])\*(z[k]-z[i]);

double zn = (x[j]-x[i])\*(y[k]-y[i]) - (x[k]-x[i])\*(y[j]-y[i]);

bool flag = zn < 0;

for (int m = 0; flag && m < n; m++)

flag = flag && ((x[m]-x[i])\*xn +

(y[m]-y[i])\*yn +

(z[m]-z[i])\*zn <= 0);

if (flag) ret.push\_back(triple(i, j, k));

}

}

}

return ret;

}

int main() {

T xs[]={0, 0, 1, 0.9};

T ys[]={0, 1, 0, 0.9};

vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);

vector<triple> tri = delaunayTriangulation(x, y);

//expected: 0 1 3

// 0 3 2

int i;

for(i = 0; i < tri.size(); i++)

printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);

return 0;

}

FFT\_new.cpp 15/27

struct cpx {

cpx(){}

cpx(double aa):a(aa){}

cpx(double aa, double bb):a(aa),b(bb){}

double a;

double b;

double modsq(void) const { return a \* a + b \* b; }

cpx bar(void) const { return cpx(a, -b); }

};

cpx operator +(cpx a, cpx b) { return cpx(a.a + b.a, a.b + b.b); }

cpx operator \*(cpx a, cpx b) { return cpx(a.a \* b.a - a.b \* b.b, a.a \* b.b + a.b \* b.a); }

cpx operator /(cpx a, cpx b) {

cpx r = a \* b.bar();

return cpx(r.a / b.modsq(), r.b / b.modsq());

}

cpx EXP(double theta) { return cpx(cos(theta),sin(theta)); }

const double two\_pi = 4 \* acos(0);

// in: input array

// out: output array

// step: {SET TO 1} (used internally)

// size: length of the input/output {MUST BE A POWER OF 2}

// dir: either plus or minus one (direction of the FFT)

// RESULT: out[k] = \sum\_{j=0}^{size - 1} in[j] \* exp(dir \* 2pi \* i \* j \* k / size)

void FFT(cpx \*in, cpx \*out, int step, int size, int dir) {

if(size < 1) return;

if(size == 1) {

out[0] = in[0];

return;

}

FFT(in, out, step \* 2, size / 2, dir);

FFT(in + step, out + size / 2, step \* 2, size / 2, dir);

for(int i = 0 ; i < size / 2 ; i++) {

cpx even = out[i];

cpx odd = out[i + size / 2];

out[i] = even + EXP(dir \* two\_pi \* i / size) \* odd;

out[i + size / 2] = even + EXP(dir \* two\_pi \* (i + size / 2) / size) \* odd;

}

}

// Usage:

// f[0...N-1] and g[0..N-1] are numbers

// Want to compute the convolution h, defined by

// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).

// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.

// Let F[0...N-1] be FFT(f), and similarly, define G and H.

// The convolution theorem says H[n] = F[n]G[n] (element-wise product).

// To compute h[] in O(N log N) time, do the following:

// 1. Compute F and G (pass dir = 1 as the argument).

// 2. Get H by element-wise multiplying F and G.

// 3. Get h by taking the inverse FFT (use dir = -1 as the argument)

// and \*dividing by N\*. DO NOT FORGET THIS SCALING FACTOR.

int main() {

printf("If rows come in identical pairs, then everything works.\n");

cpx a[8] = {0, 1, cpx(1,3), cpx(0,5), 1, 0, 2, 0};

cpx b[8] = {1, cpx(0,-2), cpx(0,1), 3, -1, -3, 1, -2};

cpx A[8];

cpx B[8];

FFT(a, A, 1, 8, 1);

FFT(b, B, 1, 8, 1);

for(int i = 0 ; i < 8 ; i++)

printf("%7.2lf%7.2lf", A[i].a, A[i].b);

printf("\n");

for(int i = 0 ; i < 8 ; i++)

{

cpx Ai(0,0);

for(int j = 0 ; j < 8 ; j++)

Ai = Ai + a[j] \* EXP(j \* i \* two\_pi / 8);

printf("%7.2lf%7.2lf", Ai.a, Ai.b);

}

printf("\n");

cpx AB[8];

for(int i = 0 ; i < 8 ; i++)

AB[i] = A[i] \* B[i];

cpx aconvb[8];

FFT(AB, aconvb, 1, 8, -1);

for(int i = 0 ; i < 8 ; i++)

aconvb[i] = aconvb[i] / 8;

for(int i = 0 ; i < 8 ; i++)

printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);

printf("\n");

for(int i = 0 ; i < 8 ; i++)

{

cpx aconvbi(0,0);

for(int j = 0 ; j < 8 ; j++)

aconvbi = aconvbi + a[j] \* b[(8 + i - j) % 8];

printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);

}

printf("\n");

}

## Simplex

// Two-phase simplex algorithm for solving linear programs of the form

// maximize c^T x

// subject to Ax <= b

// x >= 0

// INPUT: A -- an m x n matrix

// b -- an m-dimensional vector

// c -- an n-dimensional vector

// x -- a vector where the optimal solution will be stored

// OUTPUT: value of the optimal solution (infinity if unbounded

// above, nan if infeasible)

// To use this code, create an LPSolver object with A, b, and c as

// arguments. Then, call Solve(x).

#include <limits>

typedef long double DOUBLE;

typedef vector<DOUBLE> VD;

typedef vector<VD> VVD;

typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {

int m, n;

VI B, N;

VVD D;

LPSolver(const VVD &A, const VD &b, const VD &c) :

m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {

for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];

for (int i = 0; i < m; i++) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i]; }

for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }

N[n] = -1; D[m+1][n] = 1;

}

void Pivot(int r, int s) {

for (int i = 0; i < m+2; i++) if (i != r)

for (int j = 0; j < n+2; j++) if (j != s)

D[i][j] -= D[r][j] \* D[i][s] / D[r][s];

for (int j = 0; j < n+2; j++) if (j != s) D[r][j] /= D[r][s];

for (int i = 0; i < m+2; i++) if (i != r) D[i][s] /= -D[r][s];

D[r][s] = 1.0 / D[r][s];

swap(B[r], N[s]);

}

bool Simplex(int phase) {

int x = phase == 1 ? m+1 : m;

while (true) {

int s = -1;

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

if (phase == 2 && N[j] == -1) continue;

if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;

}

if (D[x][s] >= -EPS) return true;

int r = -1;

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

if (D[i][s] <= 0) continue;

if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||

D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;

}

if (r == -1) return false;

Pivot(r, s);

}

}

DOUBLE Solve(VD &x) {

int r = 0;

for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;

if (D[r][n+1] <= -EPS) {

Pivot(r, n);

if (!Simplex(1) || D[m+1][n+1] < -EPS) return -numeric\_limits<DOUBLE>::infinity();

for (int i = 0; i < m; i++) if (B[i] == -1) {

int s = -1;

for (int j = 0; j <= n; j++)

if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;

Pivot(i, s);

}

}

if (!Simplex(2)) return numeric\_limits<DOUBLE>::infinity();

x = VD(n);

for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];

return D[m][n+1];

}

};

int main() {

const int m = 4;

const int n = 3;

DOUBLE \_A[m][n] = {

{ 6, -1, 0 },

{ -1, -5, 0 },

{ 1, 5, 1 },

{ -1, -5, -1 }

};

DOUBLE \_b[m] = { 10, -4, 5, -5 };

DOUBLE \_c[n] = { 1, -1, 0 };

VVD A(m);

VD b(\_b, \_b + m);

VD c(\_c, \_c + n);

for (int i = 0; i < m; i++) A[i] = VD(\_A[i], \_A[i] + n);

LPSolver solver(A, b, c);

VD x;

DOUBLE value = solver.Solve(x);

cerr << "VALUE: "<< value << endl;

cerr << "SOLUTION:";

for (size\_t i = 0; i < x.size(); i++) cerr << " " << x[i];

cerr << endl;

return 0;

}

## KDTree

// --------------------------------------------------------------------------

// A straightforward, but probably sub-optimal KD-tree implmentation that's

// probably good enough for most things (current it's a 2D-tree)

// - constructs from n points in O(n lg^2 n) time

// - handles nearest-neighbor query in O(lg n) if points are well distributed

// - worst case for nearest-neighbor may be linear in pathological case

// --------------------------------------------------------------------------

#include <limits>

typedef long long ntype;

const ntype sentry = numeric\_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D

struct point {

ntype x, y;

point(ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}

};

bool operator==(const point &a, const point &b) {

return a.x == b.x && a.y == b.y; }

bool on\_x(const point &a, const point &b) {

return a.x < b.x; }

bool on\_y(const point &a, const point &b) {

return a.y < b.y; }

ntype pdist2(const point &a, const point &b) {

ntype dx = a.x-b.x, dy = a.y-b.y;

return dx\*dx + dy\*dy;

}

struct bbox {

ntype x0, x1, y0, y1;

bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

void compute(const vector<point> &v) {

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

x0 = min(x0, v[i].x); x1 = max(x1, v[i].x);

y0 = min(y0, v[i].y); y1 = max(y1, v[i].y);

}

}

ntype distance(const point &p) {

if (p.x < x0) {

if (p.y < y0) return pdist2(point(x0, y0), p);

else if (p.y > y1) return pdist2(point(x0, y1), p);

else return pdist2(point(x0, p.y), p);

}

else if (p.x > x1) {

if (p.y < y0) return pdist2(point(x1, y0), p);

else if (p.y > y1) return pdist2(point(x1, y1), p);

else return pdist2(point(x1, p.y), p);

}

else {

if (p.y < y0) return pdist2(point(p.x, y0), p);

else if (p.y > y1) return pdist2(point(p.x, y1), p);

else return 0;

}

}

};

struct kdnode {

bool leaf; // true if this is a leaf node (has one point)

point pt; // the single point of this is a leaf

bbox bound; // bounding box for set of points in children

kdnode \*first, \*second; // two children of this kd-node

kdnode() : leaf(false), first(0), second(0) {}

~kdnode() { if (first) delete first; if (second) delete second; }

// intersect a point with this node (returns squared distance)

ntype intersect(const point &p) {

return bound.distance(p); }

// recursively builds a kd-tree from a given cloud of points

void construct(vector<point> &vp) {

// compute bounding box for points at this node

bound.compute(vp);

// if we're down to one point, then we're a leaf node

if (vp.size() == 1) {

leaf = true;

pt = vp[0];

}

else {

// split on x if the bbox is wider than high (not best heuristic...)

if (bound.x1-bound.x0 >= bound.y1-bound.y0)

sort(vp.begin(), vp.end(), on\_x);

// otherwise split on y-coordinate

else

sort(vp.begin(), vp.end(), on\_y);

// divide by taking half the array for each child

// (not best performance if many duplicates in the middle)

int half = vp.size()/2;

vector<point> vl(vp.begin(), vp.begin()+half);

vector<point> vr(vp.begin()+half, vp.end());

first = new kdnode(); first->construct(vl);

second = new kdnode(); second->construct(vr);

}

}

};

// simple kd-tree class to hold the tree and handle queries

struct kdtree {

kdnode \*root;

// constructs a kd-tree from a points (copied here, as it sorts them)

kdtree(const vector<point> &vp) {

vector<point> v(vp.begin(), vp.end());

root = new kdnode();

root->construct(v);

}

~kdtree() { delete root; }

// recursive search method returns squared distance to nearest point

ntype search(kdnode \*node, const point &p) {

if (node->leaf) {

// commented special case tells a point not to find itself

// if (p == node->pt) return sentry;

// else

return pdist2(p, node->pt);

}

ntype bfirst = node->first->intersect(p);

ntype bsecond = node->second->intersect(p);

// choose the side with the closest bounding box to search first

// (note that the other side is also searched if needed)

if (bfirst < bsecond) {

ntype best = search(node->first, p);

if (bsecond < best)

best = min(best, search(node->second, p));

return best;

} else {

ntype best = search(node->second, p);

if (bfirst < best)

best = min(best, search(node->first, p));

return best;

}

}

// squared distance to the nearest

ntype nearest(const point &p) {

return search(root, p); }

};

int main(){

vector<point> vp;

for (int i = 0; i < 100000; ++i) {

vp.push\_back(point(rand()%100000, rand()%100000));

}

kdtree tree(vp);

for (int i = 0; i < 10; ++i) {

point q(rand()%100000, rand()%100000);

cout << "Closest squared distance to (" << q.x << ", " << q.y << ")"

<< " is " << tree.nearest(q) << endl;

}

}

// 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

// Other primes:

// The largest prime smaller than 10 is 7.

// The largest prime smaller than 100 is 97.

// The largest prime smaller than 1000 is 997.

// The largest prime smaller than 10000 is 9973.

// The largest prime smaller than 100000 is 99991.

// The largest prime smaller than 1000000 is 999983.

// The largest prime smaller than 10000000 is 9999991.

// The largest prime smaller than 100000000 is 99999989.

// The largest prime smaller than 1000000000 is 999999937.

// The largest prime smaller than 10000000000 is 9999999967.

// The largest prime smaller than 100000000000 is 99999999977.

// The largest prime smaller than 1000000000000 is 999999999989.

// The largest prime smaller than 10000000000000 is 9999999999971.

// The largest prime smaller than 100000000000000 is 99999999999973.

// The largest prime smaller than 1000000000000000 is 999999999999989.

// The largest prime smaller than 10000000000000000 is 9999999999999937.

// The largest prime smaller than 100000000000000000 is 99999999999999997.

// The largest prime smaller than 1000000000000000000 is 999999999999999989.

## 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 "?";

}

## Hamiltonian Path

**DP over subsets**  
Consider a set of elements numbered from 0 to *N* - 1. Each subset of this set can be encoded by a sequence of *N* bits (we will call this sequence "a mask"). The *i*-th element belongs to the subset if and only if the *i*-th bit of the mask equals 1. For instance, the mask00010011 means that the subset of the set [0... 7] consists of elements 0, 1 and 4. There are totally 2*N* masks, and so 2*N* subsets. Each mask is in fact an integer number written in binary notation.  
  
The method is to assign a value to each mask (and, therefore, to each subset) and compute the values for new masks using already computed values. As a rule, to find the value for a subset *A* we remove an element in every possible way and use values for obtained subsets *A*'1, *A*'2, ... , *A*'*k* to compute the value for *A*. This means that the values for *Ai*' must have been computed already, so we need to establish an ordering in which masks will be considered. It's easy to see that the natural ordering will do: go over masks in increasing order of corresponding numbers.  
  
We will use the following notation:  
*bit*(*i*, *mask*) - the *i*-th bit of *mask*  
*count*(*mask*) - the number of non-zero bits in *mask*  
*first*(*mask*) - the number of the lowest non-zero bit in *mask*  
(*a*?*b*: *c*) - returns *b* if *a* holds, or *c* otherwise.  
The elements of our set will be vertices of the graph. For the sake of simplicity we'll assume that the graph is undirected. Modification of the algorithms for directed graphs is left as an exercise for the reader.  
**1. Search for the shortest Hamiltonian walk**  
Let the graph *G* = (*V*, *E*) have *n* vertices, and each edge http://codeforces.ru/renderer/fb97fc98861a658734b29d8cc24d171bea427581.png have a weight *d*(*i*, *j*). We want to find a Hamiltonian walk for which the sum of weights of its edges is minimal.  
  
Let *dp*[*mask*][*i*] be the length of the shortest Hamiltonian walk in the subgraph generated by vertices in *mask*, that ends in the vertex*i*.  
  
The DP can be calculated by the following formulas:  
*dp*[*mask*][*i*] = 0, if *count*(*mask*) = 1 and *bit*(*i*, *mask*) = 1;  
http://codeforces.ru/renderer/a0015e8e80f8e4593f8a33fb454f299ea7b3826e.png, if *count*(*mask*) > 1 and *bit*(*i*, *mask*) = 1;  
*dp*[*mask*][*i*] = ∞ in other cases.  
  
Now the desired minimal length is http://codeforces.ru/renderer/08eb43b83b4ebb0f7c8ed636fc97b5e75b147027.png. If *pmin* = ∞, then there is no Hamiltonian walk in the graph. Otherwise it's easy to recover the walk itself. Let the minimal walk end in the vertex *i*. Then the vertex *j* ≠ *i*, for which http://codeforces.ru/renderer/9ca73bb90455f811deac9b96bd79fc78cbbff63a.png, is the previous vertex in the path. Now remove *i* from the set and find the vertex previous to *j* in the same way. Continuing this process until only one vertex is left, we'll find the whole Hamiltonian walk.  
  
This solution requires *O*(2*nn*) of memory and *O*(2*nn*2) of time.  
**2. Finding the number of Hamiltonian walks**  
Let the graph *G* = (*V*, *E*) be unweighted. We'll modify the previous algorithm. Let *dp*[*mask*][*i*] be the number of Hamiltonian walks on the subset *mask*, which end in the vertex *i*. The DP is rewritten in the following way:  
*dp*[*mask*][*i*] = 1, if *count*(*mask*) = 1 and *bit*(*i*, *mask*) = 1;  
http://codeforces.ru/renderer/fc2ed20714a99450e1772b3c135a7df97576e255.png, if *count*(*mask*) > 1 and *bit*(*i*, *mask*) = 1;  
*dp*[*mask*][*i*] = 0 in other cases.  
  
The answer is http://codeforces.ru/renderer/4c5a229855b809f2f3171898869f4d703c5e73ca.png.  
  
This solution requires *O*(2*nn*) of memory and *O*(2*nn*2) of time.  
  
**3. Finding the number of simple paths**  
Calculate the DP from the previous paragraph. The answer is http://codeforces.ru/renderer/f557b3b2119d98abd7e4bb5acc596887a5f7d3be.png. The coefficient 1 / 2 is required because each simple path is considered twice - in both directions. Also note that only paths of positive length are taken into account. You can add *n* zero-length paths, of course.  
This solution requires *O*(2*nn*) of memory and *O*(2*nn*2) of time.  
**4. Check for existence of Hamiltonian walk**  
We can use solution 2 replacing the sum with bitwise OR. *dp*[*mask*][*i*] will contain a boolean value - whether there exists a Hamiltonian walk over the subset *mask* which ends in the vertex *i*. DP is the following:  
*dp*[*mask*][*i*] = 1, if *count*(*mask*) = 1 and *bit*(*i*, *mask*) = 1;  
http://codeforces.ru/renderer/c2478038ca38c74b2cfd467c42e651b088e0f05f.png, if *count*(*mask*) > 1 and *bit*(*i*, *mask*) = 1;  
*dp*[*mask*][*i*] = 0 in other cases.  
  
This solution, like solution 2, requires *O*(2*nn*) of memory and *O*(2*nn*2) of time. It can be improved in the following way.  
Let *dp*'[*mask*] be the mask of the subset consisting of those vertices *j* for which there exists a Hamiltonian walk over the subset *mask*ending in *j*. In other words, we 'compress' the previous DP: *dp*'[*mask*] equals http://codeforces.ru/renderer/7d4593d06955f522a292beb4f52fd74bac547795.png. For the graph *G* write out *n* masks *Mi*, which give the subset of vertices incident to the vertex *i*. That is, http://codeforces.ru/renderer/082d67972740fc35c06193e688cd89b618cd5917.png.  
DP will be rewritten in the following way:  
*dp*'[*mask*] = 2*i*, if *count*(*mask*) = 1 and *bit*(*i*, *mask*) = 1;  
http://codeforces.ru/renderer/2fc99852d3d56bb20213a5304fab1d7cd3726565.png, if *count*(*mask*) > 1;  
*dp*'[*mask*] = 0 in other cases.  
Pay special attention to the expression http://codeforces.ru/renderer/b357031f3a225140e117d57f9cc738b0ef3e7df1.png. The first part of the expression is the subset of vertices *j*, for which there exists a Hamiltonian walk over the subset *mask* minus vertex *i*, ending in *j*. The second part of the expression is the set of vertices incident to *i*. If these subsets have non-empty intersection (their bitwise AND is non-zero), then it's easy to see that there exists a Hamiltonian walk in *mask* ending in the vertex *i*.  
  
The final test is to compare *dp*[2*n* - 1] to 0.  
This solution uses *O*(2*n*) of memory and *O*(2*nn*) of time.  
**5. Finding the shortest Hamiltonian cycle**  
Since we don't care at which vertex the cycle starts, assume that it starts at 0. Now use solution 1 for the subset of vertices, changing the formulas in the following way:  
*dp*[1][0] = 0;  
http://codeforces.ru/renderer/a0015e8e80f8e4593f8a33fb454f299ea7b3826e.png, if *i* > 0, *bit*(0, *mask*) = 1 and *bit*(*i*, *mask*) = 1;  
*dp*[*mask*][*i*] = ∞ in other cases.  
  
So *dp*[*mask*][*i*] contains the length of the shortest Hamiltonian walk over the subset *mask*, starting at 0 and ending at *i*.  
  
The required minimum is calculated by the formula http://codeforces.ru/renderer/f7dabda44d8bdcb97e43ed9a9dbe4a69be447ab5.png. If it equals ∞, there is no Hamiltonian cycle. Otherwise the Hamiltonian cycle can be restored by a method similar to solution 1.  
**6. Finding the number of Hamiltonian cycles**  
Using ideas from solutions 5 and 2 one can derive a DP calculating the number of Hamiltonian cycles requiring *O*(2*nn*2) of time and*O*(2*nn*) of memory.  
**7. Finding the number of simple cycles**  
Let *dp*[*mask*][*i*] be the number of Hamiltonian walks over the subset *mask*, starting at the vertex *first*(*mask*) and ending at the vertex*i*. DP looks like this:  
*dp*[*mask*][*i*] = 1, if *count*(*mask*) = 1 and *bit*(*i*, *mask*) = 1;  
http://codeforces.ru/renderer/be20d45a4a6ce0cb7230908baaafdbccf7ffd149.png, if *count*(*mask*) > 1, *bit*(*i*, *mask*) = 1 and *i* ≠ *first*(*mask*);  
*dp*[*mask*][*i*] = 0 otherwise.  
  
The answer is http://codeforces.ru/renderer/16c61a58628b3fb19238ef6db464988c35f5cc50.png.  
This solution requres *O*(2*nn*2) of time and *O*(2*nn*) of memory.  
  
**8. Checking for existence of Hamiltonian cycle**  
We can modify solution 5 and, using the trick from solution 4, obtain an algorithm requiring *O*(2*nn*) of time and *O*(2*n*) of memory.

## Factorial Modulo

int factmod (int n, int p) {

long long 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 int (res % p);

}