BFS 7

DFS 9

Binary Search 10

Dijkstra with Heap 10

Dijkstra (n2) 12

Floyed Warshal 12

Prim 13

Kruskal 14

Maximum Flow 14

Max Matching Recursively 16

Min Cost Max Flow 17

Dinic max flow 18

Strongly Connected Components 20

Kth Shortest Path 21

Topological Sort 22

Disjoint Set 23

Euler Tour 24

Extended GCD 25

Euler Totient 26

Modular Linear Equation Solver 27

Farey genrate all fractions On pairs with num. and dum. less than n 27

Continued Fractions of Rationales x=a0+( 1/ ( a1 + (1 / (a2 + ...) ) ) 27

Catalan numbers The number of distinct binary trees of n nodes 27

Prime Factors & Divisors 28

Sieve 28

nCk 28

Recursive combinations O(N^2) 29

Efficient combinations 29

-ve Base Conversion 30

SystemOfLinearEquationModed 30

Solve System of Linear Equations 31

Matrix Power 32

Integer roots for polynomial given coefficient 33

Prime power in !N 34

Geometry 34

Line Line Intersect 35

Normalize 35

Is Point On Ray 35

Is Point on Line Segment 36

Point on same side of a line 36

Get Side Sine Rule 36

Get Angle Sine Rule 36

Cosine Rule 36

Polygon Area 37

Polygon Boundary Points Count 37

Line Lattice Points Count 37

Point To Line Distance 37

Point To Line Segment Distance 37

Collinear 38

Is Point Inside Polygon 38

Polygon Centroid 39

Line Circle Intersection 39

Circle Circle Intersection 40

Three Points Circle 41

Minimum Enclosing Circle 41

Convex Hull 42

triangleAreaBH 43

triangleArea2sidesAngle 43

triangleArea2anglesSide 43

triangleArea3sides 43

triangleArea3points 43

Count Interior Points 43

Circle Point 44

Polygon Cut 44

Distance On Sphere 45

Circle Tangent 45

3D Geometry 45

3D Point 45

4x4 Transformation Matrix 47

4x4 Identity Matrix 48

3D Translation Matrix 48

3D Rotation around Z Axis Matrix 48

3D Transform coordinate system Matrix 48

3D Inverse Transform coordinate system Matrix 49

3D Get Perpendicular on two Vectors 49

3D Rotation around General line Matrix 49

Line Plane Intersection 49

Calculate the intersection of a line (not line segment) and a sphere 50

Tetrahedron centroid 50

Tetrahedron volume 51

Spherical To Cartesian Coordiantes 51

Math 51

Numerical Integration 51

Simpsons 51

Adaptive Simpsons 52

Simplex 52

Other 57

Closest Pair of Points O(N lg N) 57

CCW 58

Max Empty Rectangle 58

LIS O(N lg K) 60

RMQ 61

LCA 62

LCA on DAG 63

BIT 65

2D BIT 66

Suffix Arrays 67

O(N^2 lg N) 67

O(N (lg N)^2) 67

O(N lg N) 68

LCP 69

Suffix Tree 69

KMP 72

Rabin Karp 73

V1 73

V2 73

Aho 74

2 SAT 75

Algorithm X 76

Stable Marriage Problem 79

Bi – Connectivity 80

Bellman Ford 81

Partitioning 82

Treap 82

Expressions and Parsing 84

KD-Tree 90

FFT 92

Fraction 93

Other 94

Mimimun cycle mean 94

Ternary Search 95

Consecutive integers that sum to a given value 95

Calculating the palindrome substrings 96

Permutation Cycles (disjoint cycles) 96

Flatten rectangles 97

Letter tree 97

Letter Tree(Hashing) 98

Letter Tree(Hashing-using hashmap) 99

next\_permuatation in java 100

Permutations 101

Date 102

Solving defragmentation problem using segment trees 103

Quad Tree 105

String utilities 105

Int utilities 107

Binary Search using for loop 108

merge vector of pairs 108

Loop on all subsets of 1s for a certain number s 108

kthRoot 109

numDigits 1000 has four digits 109

Roll die 109

Time to string 110

Month names 110

Number names and fromNumTOWords and fromWordsTONum 110

st for 1 21 31, nd for 2 22, rd for 3 23 111

Return angle from hour hand to minute hand. 112

add "1234" + "56546" = "57780" givin base 112

decToBase 113

toDecimal 113

roman\_to\_int 113

int\_to\_roman 113

Joseph 114

grayCode 115

stirling1 115

stirling2 115

build\_bellNumbers 115

num\_digits\_of\_n\_combination\_k 116

fast\_Fibonaccie 116

repeating\_digits\_after\_decmial\_point\_from\_rational\_mumber 116

CONTEST STRATEGY 118

HINTS FOR THE CONTEST 118

WHY WRONG ANSWER 121

## BFS with path

int getID(char a, char b) {

return (a - 'A') \* 26 + (b - 'A');

}

string getStr(int val) {

string ret;

ret += val / 26 + 'A';

ret += val % 26 + 'A';

return ret;

}

int head[26 \* 26];

vector<pair<int, int> > edge;

int last;

// multiply m \* 2 if bidirectional edges

void init(int n, int m) {

memset(head,-1, sizeof(head[0])\*n);

last = 0;

edge.resize(m);

}

void addEdge(int f, int t) {

edge[last].second = head[f];

head[f] = last;

edge[last++].first = t;

}

struct node {

int id, par;

};

int ID;

int vis[26 \* 26];

// watch for stack overflow

void print(vector<node>& Q, int idx) {

int j = Q[idx].par;

if (j != -1) {

print(Q, j);

cout << getStr(Q[j].id) << " " << getStr(Q[idx].id) << "\n";

}

}

void bfs(int start, int goal) {

if (ID++)

cout << "\n";

vis[start] = ID;

vector<node> Q;

Q.push\_back( { start, -1 });

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

int cur = Q[i].id;

for (int j = head[cur]; j != -1; j = edge[j].second) {

int nxt = edge[j].first;

if (vis[nxt] == ID)

continue;

vis[nxt] = ID;

Q.push\_back( { nxt, i });

if (nxt == goal) {

print(Q, Q.size() - 1);

return;

}

}

}

cout << "No route\n";

}

int main() {

int m;

char fi, fj, ti, tj;

while (cin >> m) {

init(26\*26, m\*2);

while (m--) {

cin >> fi >> fj >> ti >> tj;

addEdge(getID(fi, fj), getID(ti, tj));

addEdge(getID(ti, tj), getID(fi, fj));

}

cin >> fi >> fj >> ti >> tj;

bfs(getID(fi, fj), getID(ti, tj));

}

}

## BFS

int di[8] = { 1, 1, -1, -1, 2, 2, -2, -2 };

int dj[8] = { 2, -2, 2, -2, 1, -1, 1, -1 };

int vis[256 \* 256];

int ID;

int bfs(char ci, char cj, char gi, char gj) {

ID++;

if (ci == gi && cj == gj)

return 0;

queue<int> Q;

Q.push(ci \* 256 + cj);

vis[ci \* 256 + cj] = ID;

int steps = 0;

while (Q.size()) {

int s = Q.size();

steps++;

while (s--) {

ci = Q.front() / 256;

cj = Q.front() % 256;

Q.pop();

for (int k = 0; k < 8; k++) {

char ni = ci + di[k];

char nj = cj + dj[k];

if (ni < 'a' || ni > 'h' || nj < '1' || nj > '8'

|| vis[ni \* 256 + nj] == ID)

continue;

vis[ni \* 256 + nj] = ID;

if (ni == gi && nj == gj)

return steps;

Q.push(ni \* 256 + nj);

}

}

}

return -1;

}

int main() {

char si, gi, sj, gj;

while (~scanf(" %c%c %c%c", &si, &sj, &gi, &gj)) {

printf("To get from %c%c to %c%c takes %d knight moves.\n", si, sj, gi, gj,

bfs(si, sj, gi, gj));

}

}

## BFS 01

int c, r;

char arr[101][101];

int ci[2], cj[2];

int idx;

int vis[101][101][4];

int ID;

const int OO = (int) 1e9;

int di[] = { 0, 1, 0, -1 };

int dj[] = { 1, 0, -1, 0 };

int bfs01() {

ID++;

deque<int> Q;

for (int i = 0; i < 4; i++)

Q.push\_back((ci[0] \* c + cj[0]) \* 4 + i);

int steps = 0;

while (Q.size()) {

int s = Q.size();

while (s--) {

int cur = Q.front();

Q.pop\_front();

int k = cur % 4;

cur /= 4;

int j = cur % c;

cur /= c;

int i = cur;

if (vis[i][j][k] == ID)

continue;

vis[i][j][k] = ID;

if (i == ci[1] && j == cj[1])

return steps;

int ni = i + di[k];

int nj = j + dj[k];

if (ni >= 0 && ni < r && nj >= 0 && nj < c && arr[ni][nj] != '\*') {

Q.push\_front((ni \* c + nj) \* 4 + k);

s++;

}

for (int dk = 1; dk < 4; dk += 2) {

int kk = (k + dk) % 4;

Q.push\_back((i \* c + j) \* 4 + kk);

}

}

steps++;

}

return OO;

}

int main() {

scanf("%d %d", &c, &r);

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

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

scanf(" %c", &arr[i][j]);

if (arr[i][j] == 'C')

ci[idx] = i, cj[idx++] = j;

}

cout << bfs01() << "\n";

return 0;

}

## DFS

int n, m;

int head[10001], to[40001], nxt[40001];

bool vis[10001];

int lst;

void init(int n){

memset(head, -1, n \* sizeof head[0]);

lst = 0;

}

void addEdge(int f, int t){

nxt[lst] = head[f];

head[f] = lst;

to[lst++] = t;

}

int dfs(int cur = 0){

int ret = 1;

vis[cur] = 1;

for(int i = head[cur]; i != -1; i = nxt[i]){

if(!vis[to[i]])

ret += dfs(to[i]);

}

return ret;

}

int main(){

int u, v;

cin >> n >> m;

init(n);

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

cin >> u >> v;

u--, v--;

addEdge(u,v);

addEdge(v,u);

}

memset(vis, 0, sizeof vis);

if(m + 1 != n || dfs() != n)

cout << "NO\n";

else cout << "YES\n";

}

## Binary Search

// TTTTFFFFFF

// last True

int st = 0, end = MX;

while (st < end) {

int mid = st + (end - st + 1) / 2;

if (valid(mid))

st = mid;

else

end = mid - 1;

}

cout << st << "\n";

// FFFFTTTTT

// first true

int st = 0, end = MX;

while (st < end) {

int mid = st + (end - st) / 2;

if (!valid(mid))

st = mid + 1;

else

end = mid;

}

cout << st << "\n";

## Binary Search using double

double st = 0.0, size = en - st;

for (size /= 2; size > eps; size /= 2) {

if (valid(st + size))

st += size;

}

## Dijkstra with Heap

int head[1001];

int next[4004], to[4004], cst[4004]; // edge

int last, n, m;

int dist[1001];

const int OO = (int) 1e9;

void init(int n){

memset(head, -1, n \* sizeof head[0]);

last = 0;

}

void addEdge(int f, int t, int c) {

to[last] = t;

cst[last] = c;

next[last] = head[f];

head[f] = last++;

}

int dij(int src, int sink) {

memset(dist + 1, 0x3f, n \* sizeof(dist[0]));

dist[src] = 0;

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

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

while (Q.size()) {

int cur = Q.top().second;

int d = -Q.top().first;

Q.pop();

if (d != dist[cur])

continue;

if (cur == sink)

return d;

for (int i = head[cur]; i != -1; i = next[i]) {

int t = to[i], dd = d + cst[i];

if (dist[t] > dd) {

dist[t] = dd;

Q.push(make\_pair(-dd, t));

}

}

}

return OO;

}

int main() {

int a, b, c; cin >> m >> n;

memset(head + 1, -1, n \* sizeof(head[0]));

while (m--) {

cin >> a >> b >> c;

addEdge(a, b, c); addEdge(b, a, c);

}

cout << dij(n, 1) << "\n";

return 0;

}

## Dijkstra (n2)

int cst[1001][1001];

int vis[1001], dist[1001];

int ID, n, m;

const int OO = (int) 1e9;

int dij(int src, int sink) {

memset(dist + 1, 0x3f, n \* sizeof(dist[0]));

dist[src] = 0;

ID++;

while (src != -1) {

vis[src] = ID;

if (sink == src)

return dist[src];

int nxt = -1, mn = OO;

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

if (dist[i] > dist[src] + cst[src][i])

dist[i] = dist[src] + cst[src][i];

if (vis[i] != ID && dist[i] < mn)

mn = dist[i], nxt = i;

}

src = nxt;

}

return OO;

}

int main() {

int a, b, c;

cin >> m >> n;

memset(cst, 0x3f, sizeof(cst));

while (m--) {

cin >> a >> b >> c;

cst[a][b] = cst[b][a] = min(cst[a][b], c);

}

cout << dij(n, 1) << "\n";

return 0;

}

## Floyd Warshal

**int** dist[205][205];

**int** next[205][205];

**void** **path**(**int** fr, **int** to) {

**if**(next[fr][to] == -1) {

cout << fr << " "; // beytalla3 kolo ma 3ada a5er node

**return**;

}

path(fr, next[fr][to]);

path(next[fr][to], to);

}

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

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

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

**if**(dist[i][k] + dist[k][j] < dist[i][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

next[i][j] = k;

}

## Prim with heap

int head[101];

int next[10001], to[10001]; // edge

double cst[10001]; // edge

int last, n, ID;

double dist[1001];

const int OO = (int) 1e9;

int vis[101];

void addEdge(int f, int t, double c) {

to[last] = t;

cst[last] = c;

next[last] = head[f];

head[f] = last++;

}

double prim(int src) {

ID++;

fill(dist, dist + n, 1e9);

dist[src] = 0;

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

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

double ret = 0;

while (Q.size()) {

int cur = Q.top().second;

double d = -Q.top().first;

Q.pop();

if (vis[cur] == ID)

continue;

vis[cur] = ID;

ret += d;

for (int i = head[cur]; i != -1; i = next[i]) {

int t = to[i];

double dd = cst[i];

if (dist[t] > dd) {

dist[t] = dd;

Q.push(make\_pair(-dd, t));

}

}

}

return ret;

}

double x[101], y[101];

int main() {

#ifndef ONLINE\_JUDGE

freopen("input.txt", "rt", stdin);

#endif

int t;

char\* s = "";

cin >> t;

while (t--) {

cin >> n;

memset(head, -1, n \* sizeof(head[0]));

last = 0;

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

cin >> x[i] >> y[i];

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

addEdge(i, j, hypot(x[i] - x[j], y[i] - y[j]));

addEdge(j, i, hypot(x[i] - x[j], y[i] - y[j]));

}

}

printf("%s%.2lf\n", s, prim(0)), s = "\n";

}

return 0;

}

## Prim (n2)

double cst[101][101];

double dist[101];

int vis[101];

int ID;

int n, m;

const int OO = (int) 1e9;

double prim(int src) {

++ID;

fill(dist, dist + n, OO);

dist[src] = 0;

double ret = 0;

while (src != -1) {

vis[src] = ID;

int nxt = -1;

double mn = OO;

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

if (dist[i] > cst[src][i])

dist[i] = cst[src][i];

if (vis[i] != ID && dist[i] < mn)

mn = dist[i], nxt = i;

}

if (nxt != -1)

ret += mn;

src = nxt;

}

return ret;

}

double x[101], y[101];

int main() {

#ifndef ONLINE\_JUDGE

freopen("input.txt", "rt", stdin);

#endif

int t;

char\* s = "";

cin >> t;

while (t--) {

cin >> n;

m = 0;

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

cin >> x[i] >> y[i];

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

cst[i][j] = cst[j][i] = hypot(x[i] - x[j], y[i] - y[j]);

}

}

printf("%s%.2lf\n", s, prim(0)), s = "\n";

}

return 0;

}

## Kruskal

struct disjointSet {

vector<int> par, size, rank;

int numSet;

disjointSet(int n) {

par.resize(n), size.resize(n), rank.resize(n);

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

par[i] = i, size[i] = 1;

numSet = n;

}

//Find 1

int find(int node) {

return par[node] = (par[node] == node) ? node : find(par[node]);

}

//Find 2

int operator[](int node) {

return par[node] = (par[node] == node) ? node : (\*this)[par[node]];

}

// Join 1

bool join(int x, int y) {

x = find(x);

y = find(y);

if (x == y)

return false;

if (rank[x] < rank[y])

swap(x, y);

if (rank[x] == rank[y])

rank[x]++;

size[x] += size[y];

numSet--;

par[y] = x;

return true;

}

// Join 2

bool operator()(int x, int y) {

x = (\*this)[x];

y = (\*this)[y];

if (x == y)

return false;

if (rank[x] < rank[y])

swap(x, y);

if (rank[x] == rank[y])

rank[x]++;

size[x] += size[y];

numSet--;

par[y] = x;

return true;

}

};

struct edge {

int f, t;

double c;

bool operator<(const edge& e) const {

return c < e.c;

}

};

int m, n;

edge edges[10001];

double kruskal(vector<edge>& res) {

res.clear();

double ret = 0;

disjointSet ds(n);

sort(edges, edges + m);

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

if (ds(edges[i].f, edges[i].t)) {

ret += edges[i].c;

res.push\_back(edges[i]);

}

}

return ret;

}

double x[101], y[101];

int main() {

#ifndef ONLINE\_JUDGE

freopen("input.txt", "rt", stdin);

#endif

int t;

char\* s = "";

cin >> t;

while (t--) {

cin >> n;

m = 0;

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

cin >> x[i] >> y[i];

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

edge e = { i, j, hypot(x[i] - x[j], y[i] - y[j]) };

edges[m++] = e;

}

}

vector<edge> res;

printf("%s%.2lf\n", s, kruskal(res)), s = "\n";

}

return 0;

}

## Maximum Flow

#define IN(i) ((i)<<1)

#define OUT(i) ((IN(i))+1)

#define INF 1e9

int head[5009 \* 2], vis[5009 \* 2];

vector<int> nxt, to, from;

vector<ll> cap;

int n;

int src, snk;

int ID;

void init() {

memset(head, -1, n \* (sizeof head[0]));

from = nxt = to = vector<int>();

cap.clear();

}

void addEdge(int f, int t, ll c) {

nxt.push\_back(head[f]);

head[f] = to.size();

to.push\_back(t);

from.push\_back(f);

cap.push\_back(c);

}

void addAugEdge(int f, int t, ll c) {

addEdge(f, t, c);

addEdge(t, f, 0);

}

ll dfs(int cur, ll MX) {

if (vis[cur] == ID || MX == 0)

return 0;

vis[cur] = ID;

if (cur == snk)

return MX;

for (int i = head[cur]; i != -1; i = nxt[i]) {

int t = to[i];

LL f = dfs(t, min(MX, cap[i]));

if (!f)

continue;

cap[i] -= f;

cap[i ^ 1] += f;

return f;

}

return 0;

}

ll maxFlow(){

ll ret = 0;

if (src == snk)

return INF;

ll f;

for (ID++; (f = dfs(src, INF)), f; ID++)

ret += f;

return ret;

}

bool FF(int cur) {

if(vis[cur] == ID)

return 0;

vis[cur]=ID;

if(!(cur&1))

cout<<cur/2 + 1<<" \n"[cur==2];

if(cur==snk)

return 1;

for(int i = head[cur]; i != -1 ; i = nxt[i]) {

if((i&1) || !cap[i^1])

continue;

if(FF(to[i]))

return 1;

}

return 0;

}

int main() {

int k;

int ic = 1;

while (cin >> k >> n >> ws, k || n) {

n \*= 2;

init();

addAugEdge(IN(0), OUT(0), k);

src = IN(0);

snk = IN(1);

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

if (i > 1)

addAugEdge(IN(i), OUT(i), 1);

int t;

string s;

getline(cin, s);

stringstream ss(s);

while (ss >> t) {

addAugEdge(OUT(i), IN(t-1), INF);

}

}

cout << "Case " << ic++ << ":" << endl;

if (maxFlow() < k) {

cout << "Impossible\n\n";

continue;

}

ID++;

for(int i=0;i<k;i++,vis[0]=vis[1]=vis[2]=0)

FF(IN(0));

cout<<endl;

}

return 0;

}

## Max Flow with scaling

bool dfs(int cur, LL MX) {

if (vis[cur] == ID || MX == 0)

return 0;

vis[cur] = ID;

if (cur == snk)

return 1;

for (int i = head[cur]; i != -1; i = nxt[i]) {

int t = to[i];

if (cap[i] < MX || !dfs(t, MX))

continue;

cap[i] -= MX;

cap[i ^ 1] += MX;

return 1;

}

return 0;

}

LL maxFlow() {

LL ret = 0;

if (src == snk)

return INF;

for (LL mx = 1ll << 62; mx; mx >>= 1)

for (ID++; dfs(src, mx); ID++)

ret += mx;

return ret;

}

## Max Flow with Dijkstra

int head[109];

typedef long long LL;

#define INF 1e18

vector<int> nxt, to;

vector<LL> cap;

int src, snk, n;

int ID;

void init() {

memset(head, -1, n \* (sizeof head[0]));

nxt = to = vector<int>();

cap.clear();

}

void addEdge(int f, int t, int c) {

nxt.push\_back(head[f]);

head[f] = to.size();

to.push\_back(t);

cap.push\_back(c);

}

void addAugEdge(int f, int t, int c) {

addEdge(f, t, c);

addEdge(t, f, 0);

}

LL flow[109];

typedef pair<int, int> pi;

pi parent[109];

LL Dijkstra() {

memset(flow, 0, n \* (sizeof flow[0]));

typedef pair<LL, int> pr;

priority\_queue<pr> q;

q.push(pr(INF, src));

flow[src] = INF;

while (q.size()) {

pr cur = q.top();

q.pop();

int node = cur.second;

LL f = cur.first;

if (node == snk)

break;

if (f != flow[node])

continue;

for (int i = head[node]; i != -1; i = nxt[i]) {

int t = to[i];

if (min(f, cap[i]) > flow[t]) {

flow[t] = min(f, cap[i]);

parent[t] = pi(node, i);

q.push(pr(flow[t], t));

}

}

}

LL ret = flow[snk];

for (int cur = snk; cur != src && ret; cur = parent[cur].first) {

int edgeID = parent[cur].second;

cap[edgeID] -= ret;

cap[edgeID ^ 1] += ret;

}

return ret;

}

LL maxFlow() {

LL ret = 0;

if (src == snk)

return INF;

LL f;

while (f = Dijkstra(), f)

ret += f;

return ret;

}

void addBiEdge(int f, int t, int c) {

addAugEdge(f, t, c);

addAugEdge(t, f, c);

}

int main() {

int ic = 1;

while (cin >> n, n) {

init();

int m;

cin >> src >> snk >> m;

src--, snk--;

while (m--) {

int a, b, c;

cin >> a >> b >> c;

a--, b--;

addBiEdge(a, b, c);

}

cout << "Network " << ic++

<< "\nThe bandwidth is " <<

maxFlow() << ".\n\n";

}

return 0;

}

## Max Matching Recursively: O(left \* right)

int n, m;

int lf[209], rt[209];

int head[209], nxt[209 \* 209], to[209 \* 209];

int last, ID;

int vis[209];

void init() {

last=0;

memset(head,-1,sizeof head);

memset(lf,-1,sizeof lf);

memset(rt,-1,sizeof rt);

}

void addEdge(int f, int t) {

nxt[last] = head[f];

to[last] = t;

head[f] = last++;

}

bool match(int cur) {

if (vis[cur] == ID)

return 0;

vis[cur] = ID;

for (int i = head[cur]; i != -1; i = nxt[i]) {

int t = to[i];

if (rt[t] == -1 || match(rt[t])) {

rt[t] = cur, lf[cur]=t;

return 1;

}

}

return 0;

}

int maxMatch() {

int ret = 0;

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

if (ID++, match(i))

ret++;

}

return ret;

}

int main() {

cin >> n >> m

init();

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

int a, b;

cin>>a;

while(a--) {

cin>>b;

addEdge(i, b-1);

}

}

cout<<maxMatch()<<endl;

return 0;

}

## Min Cost Max Flow

int n, src, snk; // assign values!

const int EDGEMAX = 2 \* 100 + 100 \* 100 \* 2 + 2 \* 100 + 2;

int head[209], from[EDGEMAX], nxt[EDGEMAX], to[EDGEMAX], cap[EDGEMAX],

cost[EDGEMAX];

int last, ID, vis[209];

void init() {

last = 0;

memset(head, -1, sizeof head);

}

void addEdge(int f, int t, int cst, int cp) {

nxt[last] = head[f];

to[last] = t;

cap[last] = cp;

cost[last] = cst;

from[last] = f;

head[f] = last++;

}

void addAugEdge(int f, int t, int cst, int cp) {

addEdge(f, t, cst, cp);

addEdge(t, f, -cst, 0);

}

#define INF 1e9

int dist[209], flow[209], parent[209];

int bellman() {

queue<int> Q;

ID++;

memset(dist, 0x3f, sizeof(dist[0]) \* n);

memset(flow, 0, sizeof(flow[0]) \* n);

dist[src] = 0;

flow[src] = INF;

Q.push(src);

vis[src] = ID;

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

int s = Q.size();

while (s--) {

int cur = Q.front();

Q.pop();

vis[cur] = 0;

for (int j = head[cur]; j != -1; j = nxt[j]) {

int node = to[j];

if (cap[j] && dist[node] > dist[cur] + cost[j]) {

dist[node] = dist[cur] + cost[j];

parent[node] = j;

flow[node] = min(flow[cur], cap[j]);

if (vis[node] != ID)

Q.push(node), vis[node] = ID;

}

}

}

if (Q.empty()) {

if (flow[snk]) {//found path

for (int i = snk; i != src; i = from[parent[i]]) {

int j = parent[i];

cap[j] -= flow[snk];

cap[j ^ 1] += flow[snk];

}

}

return flow[snk];

}

}

return 0;

}

pair<int,int> minCost\_maxFlow() {

int cst = 0, flw = 0;

if (src == snk)

return {0, INF};

while (bellman()) {

cst += dist[snk] \* flow[snk];

flw += flow[snk];

}

return {cst, flw};

}

int elg[128][128];

string s, t;

int main() {

cin >> s >> t;

n = (t.size() \* 2) + 2;

src = n - 2;

snk = n - 1;

init();

memset(elg, -1, sizeof elg);

for (int i = int(s.size()) - 2; i >= 0; i--)

elg[s[i]][s[i + 1]] = (i + 1) \* (i + 1);

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

addAugEdge(src, i, 0, 1);

addAugEdge(i + t.size(), snk, 0, 1);

for (int j = i + 1; j < int(t.size()); j++) {

int &a = elg[t[i]][t[j]];

if (a == -1)

continue;

addAugEdge(i, j + t.size(), a, 1);

}

}

pair<int,int> ret = minCost\_maxFlow();

cout << t.size() - ret.second << " " << ret.first << endl;

return 0;

}

## Dinic max flow

#define maxN 5002

#define maxE 30004\*2

typedef long long ct; //ct capacity type

const ct oo = 1ll << 62;

int head[maxN], headcpy[maxN], to[maxE], nxt[maxE];

ct cap[maxE];

int last;

int n, src, snk;

inline void init() {

memset(head, -1, n \* sizeof(head[0]));

last = 0;

}

inline void addEdge(int f, int t, ct cp) {

nxt[last] = head[f];

to[last] = t;

cap[last] = cp;

head[f] = last++;

}

inline void addAugEdge(int f, int t, ct c1, ct c2 = 0) {

addEdge(f, t, c1);

addEdge(t, f, c2);

}

int rank[maxN];

ct ddfs(int cur = src, ct minic = oo) {

if (cur == snk)

return minic;

for (int &i = headcpy[cur]; i != -1; i = nxt[i]) {

int t = to[i];

if (!cap[i] || rank[t] != rank[cur] + 1)

continue;

ct ret = ddfs(t, min(minic, cap[i]));

cap[i] -= ret;

cap[i ^ 1] += ret;

if (ret)

return ret;

}

return 0;

}

int Q[maxN], vis[maxN], ID = 1;

bool dbfs() {

ID++;

int Qi = 0;

Q[Qi++] = src;

vis[src] = ID;

rank[src] = 0;

for (int in = 0; in < Qi; in++) {

int cur = Q[in];

int r = rank[cur];

for (int i = head[cur]; i != -1; i = nxt[i]) {

int t = to[i];

if (!cap[i] || vis[t] == ID)

continue;

vis[t] = ID;

rank[t] = r + 1;

if (t == snk)

return 1;

Q[Qi++] = t;

}

}

return 0;

}

ct dinic() {

if (src == snk)

return oo;

ct ret = 0;

while (dbfs()) {

ct f;

memcpy(headcpy, head, n \* sizeof(head[0]));

while (f = ddfs(), f)

ret += f;

}

return ret;

}

int main() {

int m;

cin >> n >> m;

src = 0;

snk = n - 1;

init();

while (m--) {

int a, b;

ct c;

cin >> a >> b >> c;

a--;

b--;

addAugEdge(a, b, c, c);

}

cout << dinic() << "\n";

return 0;

}

## Strongly Connected Components

const int MAXN = 1001, MAXE = 1001 \* 1001;

int idx; // cur time

int head[MAXN], next[MAXE], to[MAXE], last; // adj list

int cmpIdx[MAXN]; // f(node) => its component

int stk[MAXN], stkIdx;

int dfsIdx[MAXN]; // dfs order

int lowLink[MAXN]; // f(node) => minimum time of visited node that I can reach

int VID, vis[MAXN];

int ncmp; // number of strongly connected components

int n;

bool isSrc[MAXN], isSnk[MAXN];

void init() {

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

last = 0;

}

void addEdge(int f, int t) {

next[last] = head[f];

head[f] = last;

to[last++] = t;

}

void tDFS(int cur) {

lowLink[cur] = dfsIdx[cur] = idx++;

stk[stkIdx++] = cur;

cmpIdx[cur] = -1;

vis[cur] = VID;

for (int i = head[cur]; i != -1; i = next[i]) {

int j = to[i];

if (vis[j] != VID) {

tDFS(j);

lowLink[cur] = min(lowLink[cur], lowLink[j]);

} else if (cmpIdx[j] == -1) // gray

lowLink[cur] = min(lowLink[cur], lowLink[j]);

}

if (lowLink[cur] == dfsIdx[cur]) { // component found

do {

cmpIdx[stk[--stkIdx]] = ncmp;

} while (stk[stkIdx] != cur);

ncmp++;

}

}

void SCC() {

VID++;

idx = 0;

ncmp = 0;

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

if (vis[i] != VID)

tDFS(i);

}

int main() {

while (cin >> n && n) {

init();

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

int j;

while (cin >> j, j)

addEdge(i, --j);

}

SCC();

memset(isSrc, 1, sizeof(isSrc));

memset(isSnk, 1, sizeof(isSnk));

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

for (int k = head[i]; k != -1; k = next[k]) {

int j = to[k];

int ii = cmpIdx[i], jj = cmpIdx[j];

if (ii == jj)

continue;

// To create component graph, add edge from ii to jj

isSrc[jj] = isSnk[ii] = 0;

}

}

int nSrc = accumulate(isSrc, isSrc + ncmp, 0);

int nSnk = accumulate(isSnk, isSnk + ncmp, 0);

int ret = max(nSrc, nSnk);

if (ncmp == 1)

ret = 0;

cout << nSrc << "\n" << ret << "\n";

}

return 0;

}

## Kth Shortest Path

//Nodes appear in more than one path

**struct** edge {

**int** s, e, c; //start, end, cost

**bool** **operator<**(**const** edge& e) **const** {

**return** c < e.c;

}

};

**const** **int** SIZE = 100; //max nodes number

**int** N, start, end, K; //find the k-th shortest path from start to end

**int** dist[SIZE][SIZE]; //this can be adjList instead of adjMatrix

//Returns -1 of no k-th shortest path exist between start and end

**int** **getKthShortestPath**() {

multiset<edge> pq; //first is cost and second is node

edge e = { -1, start, 0 };

pq.insert(e);

vector<**int**> reached[N]; //reached[i][j] is the cost of the j-th shortest path from start to i

**while** (!pq.empty()) {

edge e = \*pq.begin();

pq.erase(pq.begin());

**if** (reached[e.e].size() >= K)

**continue**;

reached[e.e].push\_back(e.c);

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

**if** (dist[e.e][i] == -1)

**continue**;

edge ne = { e.e, i, e.c + dist[e.e][i] };

pq.insert(ne);

}

}

//no k-th path exist between start and end

**return** reached[end].size() >= K ? reached[end].back() : -1;

}

//MAIN

memset(dist, -1, **sizeof** dist);

//set N, start, end, K, dist

**int** d = getKthShortestPath();

## Topological Sort

int n, m;

int head[101], in[101];

int nxt[10001], to[10001], last;

void addEdge(int f, int t) {

nxt[last] = head[f];

to[last] = t;

head[f] = last++;

}

vector<int> res;

void topo() {

queue<int> Q;

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

if (!in[i])

Q.push(i);

while (Q.size()) {

int cur = Q.front();

Q.pop();

res.push\_back(cur);

for (int i = head[cur]; i != -1; i = nxt[i]) {

int j = to[i];

if (!--in[j])

Q.push(j);

}

}

}

int main() {

int f, t;

while (cin >> n >> m && n) {

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

memset(in, 0, sizeof(in));

res.clear();

last = 0;

while (m--) {

cin >> f >> t;

in[t]++;

addEdge(f, t);

}

topo();

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

cout << res[i] << " \n"[i == (int) res.size() - 1];

}

return 0;

}

## Topological sort using DFS

int n, m;

int head[101], vis[101];

int nxt[10001], to[10001];

int last, id = 0;

void addEdge(int f, int t) {

nxt[last] = head[f];

to[last] = t;

head[f] = last++;

}

deque<int> res;

void topo(int idx) {

if (vis[idx] == id)

return;

vis[idx] = id;

for (int i = head[idx]; i != -1; i = nxt[i])

topo(to[i]);

res.push\_front(idx);

}

int main() {

int f, t;

while (cin >> n >> m && n) {

++id;

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

res.clear();

last = 0;

while (m--) {

cin >> f >> t;

addEdge(f, t);

}

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

topo(i);

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

cout << res[i] << " \n"[i == (int) res.size() - 1];

}

return 0;

}

## Disjoint Set

struct disjointSet {

vector<int> par, size, rank;

int numSet;

disjointSet(int n) {

par.resize(n), size.resize(n), rank.resize(n);

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

par[i] = i, size[i] = 1;

numSet = n;

}

//Find 1

int find(int node) {

return par[node] = (par[node] == node) ? node : find(par[node]);

}

//Find 2

int operator[](int node) {

return par[node] = (par[node] == node) ? node : (\*this)[par[node]];

}

// Join 1

bool join(int x, int y) {

x = find(x);

y = find(y);

if (x == y)

return false;

if (rank[x] < rank[y])

swap(x, y);

if (rank[x] == rank[y])

rank[x]++;

size[x] += size[y];

numSet--;

par[y] = x;

return true;

}

// Join 2

bool operator()(int x, int y) {

x = (\*this)[x];

y = (\*this)[y];

if (x == y)

return false;

if (rank[x] < rank[y])

swap(x, y);

if (rank[x] == rank[y])

rank[x]++;

size[x] += size[y];

numSet--;

par[y] = x;

return true;

}

};

## Euler Tour - Undirected

// Solution for Riding the Fence on USACO

// Assumes there’s always a solution

// Undirected, Euler’s Tour & Cycle

// memcpy work array!

const int MAXN = 505, MAXE = 1024 \* 2; // undirected

int head[MAXN], work[MAXN], edgecnt, ID;

int nxt[MAXE], to[MAXE], edgeIdx[MAXE], vis[MAXE];

int res[MAXE + 1], resSZ;

int resNodes[MAXE / 2 + 1], resNSZ;

void init() {

memset(head, -1, sizeof head);

memset(edgeIdx, -1, sizeof edgeIdx);

edgecnt = resSZ = resNSZ = 0;

ID++;

}

void addedge(int f, int t, int idx) {

nxt[edgecnt] = head[f];

to[edgecnt] = t;

edgeIdx[edgecnt] = idx;

head[f] = edgecnt++;

}

void addBi(int f, int t, int idx) {

addedge(f, t, idx);

addedge(t, f, idx);

}

void euler(int cur) {

for (int &ref = work[cur]; ref != -1; ) {

int eidx = edgeIdx[ref];

if (vis[ref] == ID) {

ref = nxt[ref];

continue;

}

vis[ref] = ID;

vis[ref ^ 1] = ID;

int t = to[ref];

ref = nxt[ref];

euler(t);

res[resSZ++] = eidx;

}

resNodes[resNSZ++] = cur;

// Nodes are required, push cur node to the stack here

}

pair<int,int> ed[1024];

int main() {

init();

int m;

cin >> m;

bitset<MAXN> degree;

FOR(i,0,m){

cin >> ed[i].first >> ed[i].second;

if (ed[i].first > ed[i].second)

swap (ed[i].first, ed[i].second);

degree[ed[i].first] = degree[ed[i].first] ^ 1;

degree[ed[i].second] = degree[ed[i].second] ^ 1;

}

sort(ed, ed+m);

REV(i,0,m-1)

addBi(ed[i].first, ed[i].second,i);

memcpy(work, head, sizeof head);

int si = -1, onelock = -1, negonelock = -1;

FOR(i,0,MAXN){

if(si == -1 && head[i] != -1)

si = i;

if(degree[i] == 1)

if(onelock == -1)

onelock = i;

else if(negonelock == -1)

negonelock = i;

else

onelock = -2;

}

if(onelock != -2)

if(onelock == -1) // cycle

euler(si);

else

euler(onelock); // tour

if(m != resSZ || onelock == -2)

cout << "No Sol\n";

else

REV(i,0,resNSZ - 1)

cout << resNodes[i] << "\n"; // printing nodes

return 0;

}

### Euiler Tour – Directed

// Solution for Catenyms UVa

// Directed, Euler’s Tour or Cycle

// Checks if graph is connected or not

// memcpy work!

const int N = 27;

const int E = 1001;

int head[N], edgecnt;

int nxt[E], to[E], edgeIdx[E], vis[E], ID;

int work[N];

string words[E];

int res[E], resSZ;

int degree[N];

void init() {

ID++;

memset(head, -1, sizeof head);

memset(edgeIdx, -1, sizeof edgeIdx);

memset(degree, 0, sizeof degree);

edgecnt = resSZ = 0;

}

void addedge(int f, int t, int idx) {

nxt[edgecnt] = head[f];

to[edgecnt] = t;

edgeIdx[edgecnt] = idx;

head[f] = edgecnt++;

}

void euler(int cur) {

for (int &ref = work[cur]; ref != -1; ) {

int eidx = edgeIdx[ref];

if (vis[ref] == ID) {

ref = nxt[ref];

continue;

}

vis[ref] = ID;

int t = to[ref];

ref = nxt[ref];

euler(t);

res[resSZ++] = eidx;

}

// If Nodes are required, push cur node to the stack here

}

int main() {

int t;

cin >> t;

while(t--){

init();

int n;

cin >> n;

FOR(i,0,n)

cin >> words[i];

sort(words, words + n);

int f,t;

REV(i,0,n-1){

f = words[i][0] - 'a';

t = words[i][SZ(words[i]) - 1] - 'a';

addedge(f, t, i);

degree[f]++, degree[t]--;

}

int si = -1, onelock = -1, negonelock = -1;

FOR(i,0,26){

if(si == -1 && head[i] != -1)

si = i;

if(degree[i] == 1)

if(onelock == -1)

onelock = i;

else

onelock = -2;

if(degree[i] == -1)

if(negonelock == -1)

negonelock = i;

else

onelock = -2;

if(degree[i] > 1 || degree[i] < -1)

onelock = -2;

}

if(onelock == -2){ // failed for one of the 3 reasons

cout << "\*\*\*\n";

continue;

}

memcpy(work,head,sizeof head);

euler((onelock == -1)? si: onelock);

if(resSZ != n) { // disconnected graph

cout << "\*\*\*\n";

continue;

}

REV(i,0,resSZ-1)

cout << words[res[i]] << ".\n"[i==0];

}

return 0;

}

## Extended GCD

//ax+by=gcd(a,b)

**int** **eGCD**(**int** a, **int** b, **int** &x, **int** &y) {

x = 1;

y = 0;

**int** nx = 0, ny = 1;

**int** t, r;

**while** (b) {

r = a / b;

t = a - r \* b;

a = b;

b = t;

t = x - r \* nx;

x = nx;

nx = t;

t = y - r \* ny;

y = ny;

ny = t;

}

**return** a;

}

//ax+by=c

**bool** **solveLDE**(**int** a, **int** b, **int** c, **int** &x, **int** &y, **int** &g) {

g = eGCD(a, b, x, y);

x \*= c / g;

y \*= c / g;

**return** (c % g) == 0;

}

//(a\*mi)%m=1

**int** **modInv**(**int** a, **int** m) {

**int** mi, r;

eGCD(a, m, mi, r);

**return** (mi + m) % m;

}

//(a\*x)%b=c

**bool** **solve**(ll a, ll b, ll c, ll &x) {

ll y, g;

**if** (solveLDE(a, b, c, x, y, g) && a \* x + b \* y == c) {

**if** (x < 0) {

x += (abs(x) / b) \* b;

**if** (x < 0)

x += b;

}

**return** 1;

}

**return** 0;

}

## Euler Toitent

**int** **phi**(**int** n) {

vector<**int**> p = factor(n);

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

**if** (i && p[i] == p[i - 1])

**continue**;

n /= p[i];

n \*= p[i] - 1;

}

**return** n;

}

## Modular Linear Equation Solver

returns the solutions (values of x) to the equation ax=b mod n. (n must be positive)

vector<**int**> modularLinearEquationSolver(**int** a, **int** b, **int** n){

vector<**int**> result;

triple t = extendedEuclid(a,n);

**if**(b%t.d == 0){ //if d|b

//we have solutions

**int** x0 = (t.x\*(b/t.y)) % n;

**while**(x0 < 0)x0+=n;

**for**(**int** i = 0 ; i < t.d ; i++){

**int** s = (x0 + i\*(n/t.d)) % n;

**while**(s < 0)s+=n;

result.push\_back(s);

}

}

**return** result;

}

## Farey genrate all fractions On pairs with num. and dum. less than n

//sorted

**void** **farey**(**int** n) {

// genrate all fractions On pairs with num. and dum. less than n sorted

**int** a = 0, b = 1, c = 1, d = n;

v.push\_back(make\_pair(a, b));

**while** (c < n) {

**int** k = **int**((n + b) / d), ob = b, oa = a;

a = c, b = d, c = k \* c - oa, d = k \* d - ob;

v.push\_back(make\_pair(a, b));

}

}

## Continued Fractions of Rationales x=a0+( 1/ ( a1 + (1 / (a2 + ...) ) )

//where ai is positive integer

vector<**int**> **contFract**(**int** m, **int** n) {

vector<**int**> ans;

**while** (n) {

ans.push\_back(m / n);

m %= n;

m ^= n ^= m ^= n;

}

**return** ans;

}

## Catalan numbers The number of distinct binary trees of n nodes

**long** **long** **catalan**(**int** n) {

**return** (n == 1) ?

1 :

((2 \* (n - 1) + 2) \* (2 \* (n - 1) + 1) \* catalan(n - 1))

/ ((n) \* (n + 1));

}

## Prime Factors & Divisors

**void** **factorize**(**int** n, vector<pair<**int**, **int**> > &result)

// n to get it's prime we byrga3 vector of pair for all numbers

{

result.clear();

**int** i, d = 1;

**for** (i = 2; i \* i <= n; i += d, d = 2) {

**if** (n % i == 0)

result.push\_back(make\_pair(i, 0));

**while** (n % i == 0) {

n /= i;

result.back().second++;

}

}

**if** (n != 1)

result.push\_back(make\_pair(n, 1));

**return** result;

} // worst square root(n)

vector<pair<**int**, **int**> > primeFactors;

vector<**int**> divisors;

**void** **getDivisors2**(**int** i, **int** d) { // index , divisor till now // number of divisors = powers+1 \* b3d

**if** (i == primeFactors.size()) {

divisors.push\_back(d);

**return**;

}

**for** (**int** j = 0; j <= primeFactors[i].second; j++) {

getDivisors2(i + 1, d);

d \*= primeFactors[i].first;

}

}

## Sieve

// Linear Sieve

const int N = 10000000;

int lp[N + 1];

vector<int> pr;

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

if (lp[i] == 0) {

lp[i] = i;

pr.push\_back (i);

}

for (int j=0; j<(int)pr.size() && pr[j]<=lp[i] && i\*pr[j]<=N; ++j)

lp[i \* pr[j]] = pr[j];

}

// Ordinary Sieve

const int mx = 1000005;

bool np[mx];

int si, primes[mx];

#define isPrime(x) (!np[x] && (x & 1))

void sieve() {

si = 0;

np[0] = np[1] = 1;

for (ll i = 3; i \* i <= mx; i += 2)

if (!np[i])

for (ll j = i \* i; j < mx; j += (i \* 2))

np[j] = 1;

primes[si++] = 2;

FOR (i, 0 , mx)

if (!np[i] && i % 2)

primes[si++] = i;

}

// Sieve El Fashee5

const int siz = 100000000;

int Ktos[210], stoK[48];

ll isComposite[(siz + 209) / 210];

//bool isComposite[siz];

int nums[] = {2, 3, 5, 7};

void init() {

memset(Ktos, -1, sizeof Ktos);

int j = 0;

for (int i = 0; i < 210; i ++) {

for (auto p : nums) {

if (i % p == 0)

goto nxt;

}

Ktos[i] = j;

stoK[j ++] = i;

nxt:;

}

}

void sieve\_el\_fashee5() {

isComposite[0] = 1;

// ba2fesh el start bta3 kol block with size 210

for (int i = 0; !i || i <= siz / i; i += 210) {

for (int j = 0; j < 48; j ++) {

if (!((isComposite[i / 210] >> j) & 1)) {

int k = i + stoK[j];

for (int l = k \* k; l < siz; l += k) {

int x = Ktos[l % 210];

if (x == -1)

continue;

isComposite[l / 210] |= (1LL << x);

}

}

}

}

}

inline bool isPrime(int n) {

int x = Ktos[n % 210];

if (x == -1)

return count(nums, nums + 4, n);

return !(((isComposite[n / 210]) >> x) & 1);

}

// Segmented Sieve

bool segp[M];

int n;

bool np[M];

bool segprimes[1000005];

void sieve() {

int d = 1, s = M;

np[0] = np[1] = 1;

for (ll i = 2; i < s; i += d, d = 2) {

if (!np[i]) {

for (ll j = i \* i; j < s; j += i)

np[j] = 1;

}

}

}

ll a, b;

int c1, c2, d1, d2;

void seg\_sieve() {

mem (segprimes, 0);

for (ll p = 2; p <= sqrt(b) + 1; p++) {

if (!np[p]) {

ll st = (a + p - 1) / p;

st \*= p;

if (p > a)

st = p;

for (ll i = st == p ? st + p : st; i <= b; i += p)

segprimes[i - a] = 1;

}

}

if (a == 0)

segprimes[0] = segprimes[1] = 1;

if (a == 1)

segprimes[0] = 1;

int prv = -1, mx = 0, mn = oo;

for (ll i = a; i <= b; i++) {

if (!segprimes[i - a]) {

if (prv == -1) {

prv = i;

continue;

}

if (i - prv > mx)

mx = i - prv, c1 = i, c2 = prv;

if (i - prv < mn)

mn = i - prv, d1 = i, d2 = prv;

prv = i;

}

}

}

int main() {

sieve();

while (scanf("%lld%lld", &a, &b) != -1) {

c1 = -1, c2 = -1, d1 = 0, d2 = oo;

seg\_sieve();

if (c1 != -1)

printf("%d,%d are closest, %d,%d are most distant.\n", d2, d1, c2,

c1);

else

printf("There are no adjacent primes.\n");

}

return 0;

}

## nCk

**// Use it with large values but small difference < 10e6 (take care OVERFLOW)**

**unsigned** **long** **long** **nCr**(**unsigned** **long** **long** n, **unsigned** **long** **long** r) {

**if** (n == r)

**return** 1;

**return** nCr(n - 1, r) \* n / (n - r);

}

## Recursive combinations O(N^2)

//based on pascal's formula

**const** **int** MAX = 31;

**int** comb[MAX][MAX];

**void** **calcCombinations**() {

comb[0][0] = 1;

**for** (**int** i = 1; i <= MAX; i++) {

comb[i][0] = 1;

comb[i][i] = 1;

**for** (**int** j = 1; j < i; j++)

comb[i][j] = comb[i - 1][j] + comb[i - 1][j - 1];

}

}

## Efficient combinations

//divides by the gcd before multiplication

**long** **gcd**(**long** a, **long** b) {

**if** (a % b == 0)

**return** b;

**else**

**return** gcd(b, a % b);

}

**void** **Divbygcd**(**long**& a, **long**& b) {

**long** g = gcd(a, b);

a /= g;

b /= g;

}

**long** **C**(**int** n, **int** k) {

**if** (n < k)

**return** 0;

**long** numerator = 1, denominator = 1, toMul, toDiv, i;

**if** (k > n / 2)

k = n - k; // use smaller k

**for** (i = k; i; i--) {

toMul = n - k + i;

toDiv = i;

Divbygcd(toMul, toDiv); // always divide before multiply

Divbygcd(numerator, toDiv);

Divbygcd(toMul, denominator);

numerator \*= toMul;

denominator \*= toDiv;

}

**return** numerator / denominator;

}

## -ve Base Conversion

string **ConvertToNegativeBase**(**int** x, **int** b) {

//abs(b) is between 2, 10

**bool** sign = **false**;

**if** (b > 0 && x < 0) {

sign = **true**;

x = **abs**(x);

}

string str = get(x, b);

**if** (sign)

str = "-" + str;

**return** str;

}

## SystemOfLinearEquationModedTop of Form

#include <bits/stdc++.h>

#include <ext/hash\_map>

#include <ext/numeric>

using namespace std;

using namespace \_\_gnu\_cxx;

typedef unsigned long long ull;

typedef long long ll;

typedef vector<int> vi;

typedef vector<vector<int> > vvi;

typedef pair<int, int> pii;

const int OO = (int) 2e9;

const double eps = 1e-9;

int p;

typedef vector<vector<int> > matrix;

enum sol {

NOSOL, UNIQUE, INF

};

struct mul {

int operator()(const int &x, const int &y) const {

return (x \* ll(y)) % p;

}

};

int identity\_element(const mul &x) {

return 1;

}

inline int dcmp(const int &x, const int &y) {

if (x == y)

return 0;

return (x < y) \* -2 + 1;

}

inline bool isZero(const vector<int> &v) {

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

if (dcmp(v[i], 0) != 0) // v[i] != 0 in parallel universe

return 0;

return 1;

}

inline void divideRow(vector<int>&v, const int d) {

int tmp = power(d, p - 2, mul()); // fermat's theorem (mod inverse)

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

v[i] \*= tmp, v[i] %= p;

}

inline void makeZero(vector<int> &v, vector<int> &u, int idx) {

int tmp = p - v[idx];

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

v[i] += (tmp \* u[i]) % p, v[i] %= p;

}

inline int nextZero(matrix &mat, int idx) {

for (int i = idx; i < (int) mat.size(); i++) {

if (dcmp(mat[i][idx], 0) != 0)

return i;

}

return -1;

}

void print(const matrix &mat) {

cout << flush;

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

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

cout << mat[i][j] << " ";

}

cout << "\n" << flush;

}

cout << "\n" << flush;

}

sol gauss(matrix &mat) {

sol ret = UNIQUE;

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

if (isZero(mat[i])) {

if (dcmp(mat[i].back(), 0) != 0)

return NOSOL;

mat[i].swap(mat.back());

mat.pop\_back();

i--;

continue;

}

int p = nextZero(mat, i);

if (p == -1) {

ret = INF;

continue;

}

if (p != i)

mat[i].swap(mat[p]); // O(1)

divideRow(mat[i], mat[i][i]);

// print(mat);

for (int j = 0; j < (int) mat.size(); j++) {

if (i == j || dcmp(mat[j][i], 0) == 0)

continue;

makeZero(mat[j], mat[i], i);

}

// print(mat);

}

if (mat.empty() || mat.size() < mat[0].size() - 1)

ret = INF;

return ret;

}

int main() {

std::ios\_base::sync\_with\_stdio(false);

#ifndef ONLINE\_JUDGE

freopen("in.txt", "rt", stdin);

// freopen("out.txt","wt",stdout);

#endif

int t;

string str;

cin >> t;

while (t--) {

cin >> p >> str;

matrix mat(str.size(), vector<int>(str.size() + 1));

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

mat[i].back() = str[i] == '\*' ? 0 : str[i] - 'a' + 1;

for (int j = 0; j < (int) str.size(); j++)

mat[i][j] = power(i + 1, j, mul());

}

gauss(mat);

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

cout << mat[i].back() << " \n"[i == (int) mat.size() - 1];

}

return 0;

}

## Solve System of Linear Equations (Gaussian)

using namespace \_\_gnu\_cxx;

typedef unsigned long long ull;

typedef long long ll;

typedef vector<int> vi;

typedef pair<int, int> pii;

const int OO = (int) 2e9;

const double eps = 1e-9;

typedef vector<vector<double> > matrix;

enum sol {

NOSOL, UNIQUE, INF

};

inline int dcmp(const double &x, const double &y) {

if (fabs(x - y) < eps)

return 0;

return (x < y) \* -2 + 1;

}

inline bool isZero(const vector<double> &v, vector<int> &cols) {

for (int j = 0; j < (int) cols.size() - 1; j++){

int i = cols[j];

if (dcmp(v[i], 0.0) != 0) // v[i] != 0 in parallel universe

return 0;

}

return 1;

}

inline void divideRow(vector<double>&v, const double d) {

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

v[i] /= d;

}

inline void makeZero(vector<double> &v, vector<double> &u, int idx) {

double tmp = -v[idx];

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

v[i] += tmp \* u[i];

}

inline int nextZero(matrix &mat, int i, int idx) {

for (; i < (int) mat.size(); i++) {

if (dcmp(mat[i][idx], 0.0) != 0)

return i;

}

return -1;

}

sol gauss(matrix &mat) {

sol ret = UNIQUE;

vector<int> cols;

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

cols.push\_back(i);

}

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

if (isZero(mat[i], cols)) {

if (dcmp(mat[i].back(), 0.0) != 0)

return NOSOL;

mat[i].swap(mat.back());

mat.pop\_back();

i--;

continue;

}

int p = nextZero(mat,i, cols[i]);

if (p == -1) {

ret = INF;

cols.erase(cols.begin()+i);

i--;

continue;

}

if (p != i)

mat[i].swap(mat[p]); // O(1)

// divideRow(mat[i], mat[i][i]);

divideRow(mat[i], mat[i][cols[i]]);

// print(mat);

for (int j = 0; j < (int) mat.size(); j++) {

// if (i == j || dcmp(mat[j][i], 0.0) == 0)

if (i == j || dcmp(mat[j][cols[i]], 0.0) == 0)

continue;

makeZero(mat[j], mat[i], cols[i]);

}

// print(mat);

}

// if (mat.empty() || mat.size() < mat[0].size() - 1)

if (mat.empty() || mat.size() < cols.size() - 1)

ret = INF;

return ret;

}

ull arr[1501];

int main() {

std::ios\_base::sync\_with\_stdio(false);

int t;

cin >> t;

while (t--) {

matrix mat(7, vector<double>(8));

for (int i = 0; i < 1500; i++) {

cin >> arr[i];

if (i < 7) {

mat[i].back() = arr[i];

for (int j = 0; j < 7; j++)

mat[i][j] = pow(i + 1, j);

}

}

if (gauss(mat) == UNIQUE) {

for (int i = 0; i < 1500; i++) {

ull sum = 0;

for (int j = 0; j < 7; j++) {

ull x = power((ull) (i + 1), (ull) (j));

ull a = round(mat[j].back());

if (a < 0 || a > 1000)

goto BARRA;

if (a > ULLONG\_MAX / x)

goto BARRA;

x \*= a;

if (sum > ULLONG\_MAX - x)

goto BARRA;

sum += x;

}

if (sum != arr[i])

goto BARRA;

}

for (int i = 0; i < 7; i++)

printf("%.0lf%c", mat[i].back(), " \n"[i == 6]);

} else

BARRA: puts("This is a smart sequence!");

}

return 0;

}

## Solve System of Linear Equations (Integer values)

using namespace \_\_gnu\_cxx;

typedef unsigned long long ull;

typedef long long ll;

typedef vector<int> vi;

typedef vector<vector<int> > vvi;

typedef pair<int, int> pii;

const int OO = (int) 2e9;

const double eps = 1e-9;

typedef vector<vector<ll> > matrix;

enum sol {

NOSOL, UNIQUE, INF

};

inline int dcmp(const ll &x, const ll &y) {

if (x == y)

return 0;

return (x < y) \* -2 + 1;

}

inline bool isZero(const vector<ll> &v) {

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

if (dcmp(v[i], 0) != 0) // v[i] != 0 in parallel universe

return 0;

return 1;

}

inline void divideRow(vector<ll>&v, const ll d) {

// int tmp = power(d, p - 2, mul()); // fermat's theorem (mod inverse)

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

v[i] /= d;

}

inline void makeZero(vector<ll> &v, vector<ll> &u, int idx) {

int tmp1 = -v[idx], tmp2 = u[idx];

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

v[i] = v[i] \* tmp2 + u[i] \* tmp1;

}

inline int nextZero(matrix &mat, int idx) {

for (int i = idx; i < (int) mat.size(); i++) {

if (dcmp(mat[i][idx], 0) != 0)

return i;

}

return -1;

}

void print(const matrix &mat) {

cout << flush;

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

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

cout << mat[i][j] << " ";

}

cout << "\n" << flush;

}

cout << "\n" << flush;

}

sol gauss(matrix &mat) {

sol ret = UNIQUE;

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

if (isZero(mat[i])) {

if (dcmp(mat[i].back(), 0) != 0)

return NOSOL;

mat[i].swap(mat.back());

mat.pop\_back();

i--;

continue;

}

int p = nextZero(mat, i);

if (p == -1) {

ret = INF;

continue;

}

if (p != i)

mat[i].swap(mat[p]); // O(1)

ll g = 0;

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

g = \_\_gcd(g, mat[i][j]);

divideRow(mat[i], g);

// print(mat);

for (int j = 0; j < (int) mat.size(); j++) {

if (i == j || dcmp(mat[j][i], 0) == 0)

continue;

makeZero(mat[j], mat[i], i);

}

// print(mat);

}

if (mat.empty() || mat.size() < mat[0].size() - 1)

ret = INF;

return ret;

}

void factorize(int n, vector<ll>& b, vector<ll>& pr) {

b.clear();

pr.clear();

for (int i = 2; i <= n / i; i += (1 + (i & 1))) {

int cnt = 0;

while (n % i == 0)

cnt++, n /= i;

if (cnt)

b.push\_back(i), pr.push\_back(cnt);

}

if (n > 1)

b.push\_back(n), pr.push\_back(1);

}

int arr[22];

vector<ll> prime[22], power[22];

vector<ll> A, B;

int main() {

std::ios\_base::sync\_with\_stdio(false);

int t, tt = 1, n, q, num, den, Bnum, Bden;

cin >> t;

while (t--) {

cin >> n;

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

cin >> arr[i];

sort(arr, arr + n);

reverse(arr, arr + n--);

map<ll, int> primes;

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

arr[i] /= arr[n];

factorize(arr[i], prime[i], power[i]);

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

primes[prime[i][j]];

}

cin >> q;

int cnt = 0;

for (map<ll, int>::iterator it = primes.begin(); it != primes.end(); it++)

it->second = cnt++; // map primes to specific index

printf("Scenario #%d:\n", tt++);

while (q--) {

cin >> num >> den;

Bnum = num, Bden = den;

int g = \_\_gcd(num, den);

num /= g, den /= g;

if (num == den)

goto YES;

{

matrix mat(cnt, vector<ll>(n + 1));

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

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

mat[primes[prime[i][j]]][i] = power[i][j];

factorize(num, A, B);

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

if (!primes.count(A[i]))

goto NO;

mat[primes[A[i]]].back() = B[i];

}

factorize(den, A, B);

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

if (!primes.count(A[i]))

goto NO;

mat[primes[A[i]]].back() -= B[i];

}

if (gauss(mat) == NOSOL)

goto NO;

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

ll g = 0;

for (int j = 0; j < (int) mat[i].size() - 1; j++)

g = \_\_gcd(g, mat[i][j]);

if (mat[i].back() % g != 0)

goto NO;

}

}

YES: printf("Gear ratio %d:%d can be realized.\n", Bnum, Bden);

continue;

NO: printf("Gear ratio %d:%d cannot be realized.\n", Bnum, Bden);

}

puts("");

}

return 0;

}

## Matrix Power

using namespace \_\_gnu\_cxx;

typedef unsigned long long ull;

typedef long long ll;

typedef vector<int> vi;

typedef vector<vector<int> > vvi;

typedef pair<int, int> pii;

const int OO = (int) 2e9;

const double eps = 1e-9;

const int mod = 98765431;

ll arr[50004];

int n, rounds;

struct mat {

int r, c;

vector<vector<ll> > M;

mat(int rr, int cc) :

r(rr), c(cc) {

M.resize(r, vector<ll>(c));

}

vector<ll>& operator[](int i) {

return M[i];

}

const vector<ll>& operator[](int i) const {

return M[i];

}

};

struct mul {

int r, c;

mul(int rr, int cc) {

r = rr, c = cc;

}

mat operator()(const mat&m1, const mat&m2) const {

mat ret(m1.r, m2.c);

for (int i = 0; i < m1.r; i++)

for (int j = 0; j < m2.c; j++)

for (int k = 0; k < m1.c; k++) {

ret[i][j] += (m1[i][k] \* m2[k][j]) % mod;

ret[i][j] %= mod;

}

return ret;

}

};

mat identity\_element(const mul& m) {

mat M(m.r, m.c);

for (int i = 0; i < m.r; i++)

M[i][i] = 1ll;

return M;

}

int main() {

// std::ios\_base::sync\_with\_stdio(false);

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

ll sum = 0;

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

scanf("%I64d", arr + i), sum += arr[i], sum %= mod;

mat M(2, 2);

M[0][0] = mod - 1, M[0][1] = 1;

M[1][0] = 0, M[1][1] = (n - 1) % mod;

M = power(M, rounds, mul(2, 2));

mat V(2, 1);

V[1][0] = sum;

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

V[0][0] = arr[i] % mod;

printf("%I64d\n", mul(2, 2)(M, V)[0][0]);

}

return 0;

}

## Integer roots for polynomial given coefficients

**#define** big **long** **long**

big a[100000];// the polynomial coefficients, a[0] is the coefficient of the constant term

**int** n; //the polynomial degree

big MAX\_COEFFICIENT; // the largest possible absolute value of a coefficient

**bool** **check**(big x) {

big d = 0;

**for** (**int** i = n; i >= 0; i--) {

d = d \* x + a[i];

**if** (abs(x) != 1 && abs(d) > 2 \* MAX\_COEFFICIENT)

**return** **false**;

}

**return** d == 0;

}

set<big> **getIntegerRoots**() {

set<big> res;

**if** (a[0] == 0)

res.insert(0);

**int** f = 0;

**while** (a[f] == 0)

f++; //specify constant term of the polynomial

set<big> div;

div = divisors(abs(a[f]));

//divisors of constant term, these are the possible roots

vector<big> vv(div.begin(), div.end());

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

**if** (check(vv[i]))

res.insert(vv[i]);

**if** (check(-vv[i]))

res.insert(-vv[i]);

}

**return** res;

}

//MAIN

//Set a, n, MAX\_COEFFICIENT

set<big> roots = getIntegerRoots();

## Prime power in !N

**long** **long** **count\_p\_in\_nfact**(**long** **long** p, **long** **long** n) {

**long** **long** res = 0;

**long** **long** q = p;

**while** (q <= n) {

res += n / q;

q \*= p;

}

**return** res;

}

## Geometry

typedef complex<double> point;

#define EPS 1e-9

#define OO 1e9

#define X real()

#define Y imag()

#define vec(a,b) ((b) - (a))

#define polar(r,t) ((r) \* exp(point(0, (t))))

#define angle(v) (atan2((v).Y, (v).X))

#define length(v) ((double)hypot((v).Y, (v).X))

#define lengthSqr(v) (dot(v, v))

#define dot(a,b) ((conj(a) \* (b)).real())

#define cross(a,b) ((conj(a) \* (b)).imag())

#define rotate(v,t) (polar(v, t))

#define rotateabout(v,t,a) (rotate(vec(a, v), t) + (a))

#define reflect(p,m) ((conj((p) / (m))) \* (m))

#define normalize(p) ((p) / length(p))

#define same(a,b) (lengthSqr(vec(a, b)) < EPS)

#define mid(a,b) (((a) + (b)) / point(2, 0))

#define perp(a) (point(-(a).Y, (a).X))

#define colliner pointOnLine

enum STATE {

IN, OUT, BOUNDRY

};

### Intersect

bool intersect(const point &a, const point &b,

const point &p, const point &q,point &ret) {

//handle degenerate cases (2 parallel lines, 2 identical lines, line is 1 point)

double d1 = cross(p - a, b - a);

double d2 = cross(q - a, b - a);

ret = (d1 \* q - d2 \* p) / (d1 - d2);

if(fabs(d1 - d2) > EPS) return 1;

return 0;

}

### Is Point On Ray

bool pointOnRay(const point& a, const point& b, const point& p) {

//IMP NOTE: a,b,p must be collinear

return dot(vec(a,p), vec(a,b)) > -EPS;

}

### Point On Segment

bool pointOnSegment(const point& a, const point& b, const point& p) {

if(!colliner(a,b,p)) return 0;

return pointOnRay(a, b, p) && pointOnRay(b, a, p);

}

### Point On Line

bool pointOnLine(const point& a, const point& b, const point& p) {

// degenerate case: line is a point

return fabs(cross(vec(a,b),vec(a,p))) < EPS;

}

### Point Line Dist

double pointLineDist(const point& a, const point& b, const point& p) {

// handle degenrate case: (a,b) is point

return fabs(cross(vec(a,b),vec(a,p)) / length(vec(a,b)));

}

### Point Segment Dist

double pointSegmentDist(const point &a, const point &b,const point &p){

if (dot(vec(a,b),vec(a,p)) < EPS)

return length(vec(a,p));

if (dot(vec(b,a),vec(b,p)) < EPS)

return length(vec(b,p));

return pointLineDist(a, b, p);

}

### Segment Lattice Point Count

int segmentLatticePointsCount(int x1, int y1, int x2, int y2) {

return abs(\_\_gcd(x1 - x2, y1 - y2)) + 1;

}

### Sin Rule

double sinRuleAngle(double s1, double s2, double a1) {

// Handle denom = 0

double res = s2 \* sin(a1) / s1;

if (res > 1)

res = 1;

if (res < -1)

res = -1;

return asin(res);

}

double sinRuleSide(double s1, double a1, double a2) {

// Handle denom = 0

double res = s1 \* sin(a2) / sin(a1);

return fabs(res);

}

### Cosine Rule

//get angle opposite to side a

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

// Handle denom = 0

double res = (b \* b + c \* c - a \* a) / (2 \* b \* c);

if (res > 1)

res = 1;

if (res < -1)

res = -1;

return acos(res);

}

### Triangle Area

double triangleAreaBH(double b, double h) {

return b \* h / 2;

}

double triangleArea2sidesAngle(double a, double b, double t) {

return fabs(a \* b \* sin(t) / 2);

}

double triangleArea2anglesSide(double t1, double t2,

double s) {

return fabs(s \* s \* sin(t1) \* sin(t2) / (2 \* sin(t1 + t2)));

}

double triangleArea3sides(double a, double b, double c) {

double s((a + b + c) / 2);

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

}

double triangleArea3points(const point &a, const point &b,

const point &c) {

return fabs(cross(a,b) + cross(b,c) + cross(c,a)) / 2;

}

### Pick’s Theorem

//count interior lattice points

int picksTheorem(int a, int b) {

return a - b / 2 + 1;

}

int picksTheorem(vector<point>& p) {

double area = 0;

int bound = 0;

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

int j = (i + 1) % sz(p);

area += cross(p[i], p[j]);

point v = vec(p[i], p[j]);

bound += abs(\_\_gcd((int) v.X, (int) v.Y));

}

area /= 2;

area = fabs(area);

return round(area - bound / 2 + 1);

}

### Circle Line Intersection

int circleLineIntersection(const point &p0, const point &p1,

const point &cen, double rad, point &r1, point &r2) {

// handle degenerate case if p0 == p1

double a, b, c, t1, t2;

a = dot(p1 - p0, p1 - p0);

b = 2 \* dot( p1 - p0, p0 - cen);

c = dot(p0 - cen, p0 - cen) - rad \* rad;

double det = b \* b - 4 \* a \* c;

int res;

if (fabs(det) < EPS)

det = 0, res = 1;

else if (det < 0)

res = 0;

else

res = 2;

det = sqrt(det);

t1 = (-b + det) / (2 \* a);

t2 = (-b - det) / (2 \* a);

r1 = p0 + t1 \* (p1 - p0);

r2 = p0 + t2 \* (p1 - p0);

return res;

}

### Circle Circle Intersection

int circleCircleIntersection(const point &c1, const double &r1,

const point &c2, const double &r2,

point &res1, point &res2) {

if (same(c1,c2) && fabs(r1 - r2) < EPS) {

res1 = res2 = c1;

return fabs(r1) < EPS ? 1 : OO;

}

double len = length(vec(c1,c2));

if (fabs(len - (r1 + r2)) < EPS ||

fabs(fabs(r1 - r2) - len) < EPS) {

point d, c;

double r;

if (r1 > r2)

d = vec(c1,c2), c = c1, r = r1;

else

d = vec(c2,c1), c = c2, r = r2;

res1 = res2 = normalize(d) \* r + c;

return 1;

}

if (len > r1 + r2 || len < fabs(r1 - r2))

return 0;

double a = cosRule(r2, r1, len);

point c1c2 = normalize(vec(c1,c2)) \* r1;

res1 = rotate(c1c2, a) + c1;

res2 = rotate(c1c2, -a) + c1;

return 2;

}

### Circle From 2 Points

void circle2(const point &p1, const point &p2, point &cen, double &r) {

cen = mid(p1, p2);

r = length(vec(p1, p2)) / 2;

}

### Circle From 3 Points

bool circle3(const point &p1, const point &p2, const point &p3,

point& cen, double& r) {

point m1 = mid(p1, p2);

point m2 = mid(p2, p3);

point perp1 = perp(vec(p1, p2));

point perp2 = perp(vec(p2, p3));

bool res = intersect(m1, m1 + perp1, m2, m2 + perp2, cen);

r = length(vec(cen,p1));

return res;

}

### Circle Point

STATE circlePoint(const point &cen, const double &r, const point &p) {

double lensqr = lengthSqr(vec(cen,p));

if (fabs(lensqr - r \* r) < EPS)

return BOUNDRY;

if (lensqr < r \* r)

return IN;

return OUT;

}

### Circle Tangent from Point

int tangentPoints(const point &cen, const double &r, const point &p,

point &r1, point &r2) {

STATE s = circlePoint(cen, r, p);

if (s != OUT) {

r1 = r2 = p;

return s == BOUNDRY;

}

point cp = vec(cen,p);

double h = length(cp);

double a = acos(r / h);

cp = normalize(cp) \* r;

r1 = rotate(cp,a) + cen;

r2 = rotate(cp,-a) + cen;

return 2;

}

### Minimum Enclosing Circle

//init p array with the points and ps with the number of points

//cen and rad are result circle

//you must call random\_shuffle(p,p+ps); before you call mec

#define MAXPOINTS 100000

point p[MAXPOINTS], r[3], cen;

int ps, rs;

double rad;

void mec() {

if (rs == 3) {

circle3(r[0], r[1], r[2], cen, rad);

return;

}

if (rs == 2 && ps == 0) {

circle2(r[0], r[1], cen, rad);

return;

}

if (!ps) {

cen = r[0];

rad = 0;

return;

}

ps--;

mec();

if (circlePoint(cen, rad, p[ps]) == OUT) {

r[rs++] = p[ps];

mec();

rs--;

}

ps++;

}

### Polygon Area

//to check if the points are sorted anti-clockwise or clockwise

//remove the fabs at the end and it will return -ve value if clockwise

double polygonArea(const vector<point> &p) {

double res = 0;

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

int j = (i + 1) % sz(p);

res += cross(p[i],p[j]);

}

return fabs(res) / 2;

}

### Polygon Centroid

// return the centroid point of the polygon

// The centroid is also known as the "centre of gravity" or the "center of mass". The position of the centroid

// assuming the polygon to be made of a material of uniform density.

point polygonCentroid(vector<point> &polygon) {

point res(0, 0);

double a = 0;

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

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

res.X += (polygon[i].X + polygon[j].X) \*

(polygon[i].X \* polygon[j].Y –

polygon[j].X \* polygon[i].Y);

res.Y += (polygon[i].Y + polygon[j].Y) \*

(polygon[i].X \* polygon[j].Y –

polygon[j].X \* polygon[i].Y);

a += polygon[i].X \* polygon[j].Y –

polygon[i].Y \* polygon[j].X;

}

a \*= 0.5;

res.X /= 6 \* a;

res.Y /= 6 \* a;

return res;

}

### Polygon Cut

void polygonCut(const vector<point> &p, const point &a, const point &b,

vector<point> &res) {

res.clear();

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

int j = (i + 1) % sz(p);

bool in1 = cross(vec(a,b),vec(a,p[i])) > EPS;

bool in2 = cross(vec(a,b),vec(a,p[j])) > EPS;

if (in1)

res.push\_back(p[i]);

if (in1 ^ in2) {

point r;

intersect(a, b, p[i], p[j], r);

res.push\_back(r);

}

}

}

### Convex Polygon Intersect

//assume that both are anti-clockwise

void convexPolygonIntersect(const vector<point> &p,

const vector<point> &q,

vector<point>& res) {

res = q;

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

int j = (i + 1) % sz(p);

vector<point> temp;

polygonCut(res, p[i], p[j], temp);

res = temp;

if (res.empty())

return;

}

}

### Voronoi

void voronoi(const vector<point> &pnts, const vector<point> &rect,

vector<vector<point> > &res) {

res.clear();

for (int i = 0; i < sz(pnts); i++) {

res.push\_back(rect);

for (int j = 0; j < sz(pnts); j++) {

if (j == i)

continue;

point p = perp(vec(pnts[i], pnts[j]));

point m = mid(pnts[i], pnts[j]);

vector<point> temp;

polygonCut(res.back(), m, m + p, temp);

res.back() = temp;

}

}

}

### Point In Polygon

STATE pointInPolygon(const vector<point> &p, const point &pnt) {

point p2 = pnt + point(1, 0);

int cnt = 0;

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

int j = (i + 1) % sz(p);

if (pointOnSegment(p[i], p[j], pnt))

return BOUNDRY;

point r;

if(!intersect(pnt, p2, p[i], p[j], r))

continue;

if (!pointOnRay(pnt, p2, r))

continue;

if (same(r,p[i]) || same(r,p[j]))

if (fabs(r.Y - min(p[i].Y, p[j].Y)) < EPS)

continue;

if (!pointOnSegment(p[i], p[j], r))

continue;

cnt++;

}

return cnt & 1 ? IN : OUT;

}

### Sort Anti-Clockwise

struct cmp {

point about;

cmp(point c) {

about = c;

}

bool operator()(const point &p, const point &q) const {

double cr = cross(vec(about, p), vec(about, q));

if (fabs(cr) < EPS)

return make\_pair(p.Y, p.X) < make\_pair(q.Y, q.X);

return cr > 0;

}

};

void sortAntiClockWise(vector<point> &pnts) {

point mn(1 / 0.0, 1 / 0.0);

for (int i = 0; i < sz(pnts); i++)

if (make\_pair(pnts[i].Y, pnts[i].X) < make\_pair(mn.Y, mn.X))

mn = pnts[i];

sort(all(pnts), cmp(mn));

}

### Convex Hull

void convexHull(vector<point> pnts, vector<point> &convex) {

sortAntiClockWise(pnts);

convex.clear();

convex.push\_back(pnts[0]);

if (sz(pnts) == 1)

return;

convex.push\_back(pnts[1]);

if (sz(pnts) == 2) {

if (same(pnts[0], pnts[1]))

convex.pop\_back();

return;

}

for (int i = 2; i <= sz(pnts); i++) {

point c = pnts[i % sz(pnts)];

while (sz(convex) > 1) {

point b = convex.back();

point a = convex[sz(convex) - 2];

if (cross(vec(b, a), vec(b, c)) < -EPS)

break;

convex.pop\_back();

}

if (i < sz(pnts))

convex.push\_back(pnts[i]);

}

}

### Distance On Sphere

/\* Spherical distance from longitude & latitude \*/

**double** **SphericalDist**(**double** p\_long, **double** p\_lat, **double** q\_long, **double** q\_lat,**double** r) {

**double** a = (1.0 - **cos**(q\_lat - p\_lat)) / 2, b = **cos**(p\_lat) \* **cos**(q\_lat)\* (1.0 - **cos**(q\_long - p\_long)) / 2;

**double** c = 2 \* **atan2**(**sqrt**(a + b), **sqrt**(1 - a - b));

**return** r \* c; // more accurate

}

### Circle Tangent

**bool** **circleTangent**(**const** point& c, **const** point &p, **const** **double** &r, point &p1,point &p2) {

point pc = c - p;

**double** l = length(pc);

**if** (r > l)

**return** **false**;

**double** x = **sqrt**(l \* l - r \* r),

a = **asin**(r / l), t = angle(pc), a1 = t + a,a2 = t - a;

p1 = point(**cos**(a1), **sin**(a1)) \* x + p;

p2 = point(**cos**(a2), **sin**(a2)) \* x + p;

**return** **true**;

}

# 3D Geometry

## 3D Point

**#define** EPS 1e-9

**double** ONE = 1;

**struct** point3D {

**double** v[3];

**point3D**() {

**for** (**int** i = 0; i < 3; ++i) {

**this**->v[i] = 0;

}

}

**point3D**(**double** v[3]) {

**for** (**int** i = 0; i < 3; ++i) {

**this**->v[i] = v[i];

}

}

**double**& **operator []**(**int** idx) {

**return** idx < 3 ? v[idx] : (ONE = 1);

}

**double** **operator []**(**int** idx) **const** {

**return** idx < 3 ? v[idx] : 1;

}

**double**& **x**() {

**return** v[0];

}

**double**& **y**() {

**return** v[1];

}

**double**& **z**() {

**return** v[2];

}

point3D **operator +**(**const** point3D& t) **const** {

point3D ret;

**for** (**int** i = 0; i < 3; ++i) {

ret.v[i] = v[i] + t.v[i];

}

**return** ret;

}

point3D **operator -**(**const** point3D& t) **const** {

point3D ret;

**for** (**int** i = 0; i < 3; ++i) {

ret.v[i] = v[i] - t.v[i];

}

**return** ret;

}

point3D **operator \***(**const** **double**& t) **const** {

point3D ret;

**for** (**int** i = 0; i < 3; ++i) {

ret.v[i] = v[i] \* t;

}

**return** ret;

}

point3D **operator /**(**const** **double**& t) **const** {

point3D ret;

**for** (**int** i = 0; i < 3; ++i) {

ret.v[i] = v[i] / t;

}

**return** ret;

}

**double** **Length**() {

**double** sum = 0;

**for** (**int** i = 0; i < 3; ++i) {

sum += v[i] \* v[i];

}

**return** **sqrt**(sum);

}

**double** **Dot**(**const** point3D& t) **const** {

**double** sum = 0;

**for** (**int** i = 0; i < 3; ++i) {

sum += v[i] \* t.v[i];

}

**return** sum;

}

point3D **Cross**(**const** point3D& t) **const** {

**double** arr[] = { v[1] \* t.v[2] - v[2] \* t.v[1], v[2] \* t.v[0] - v[0]

\* t.v[2], v[0] \* t.v[1] - v[1] \* t.v[0] };

**return** point3D(arr);

}

point3D **Normalize**() {

**return** point3D(v) / Length();

}

};

## 4x4 Transformation Matrix

**struct** matrix {

**double** arr[4][4];

matrix **operator \***(**const** matrix& m) **const** {

matrix ret;

**memset**(ret.arr, 0, **sizeof**(ret.arr));

**for** (**int** i = 0; i < 4; ++i) {

**for** (**int** j = 0; j < 4; ++j) {

**for** (**int** k = 0; k < 4; ++k) {

ret.arr[i][j] += arr[i][k] \* m.arr[k][j];

}

}

}

**return** ret;

}

point3D **operator \***(**const** point3D& m) **const** {

point3D ret;

**for** (**int** i = 0; i < 4; ++i) {

**for** (**int** j = 0; j < 4; ++j) {

ret[i] += arr[i][j] \* m[j];

}

}

**return** ret;

}

**double**& **operator()**(**int** i, **int** j) {

**return** arr[i][j];

}

**const** **double**& **operator()**(**int** i, **int** j) **const** {

**return** arr[i][j];

}

};

## 4x4 Identity Matrix

matrix **Identity**() {

matrix ret;

**for** (**int** i = 0; i < 4; ++i) {

**for** (**int** j = 0; j < 4; ++j) {

ret(i, j) = i == j;

}

}

**return** ret;

}

## 3D Translation Matrix

matrix **translate**(**const** point3D& v, **int** dir = 1) {

matrix ret = Identity();

**for** (**int** i = 0; i < 3; ++i) {

ret(i, 3) = v[i] \* dir;

}

**return** ret;

}

## 3D Rotation around Z Axis Matrix

matrix **rotateZ**(**double** angle) {

matrix ret = Identity();

ret(0, 0) = ret(1, 1) = **cos**(angle);

ret(0, 1) = -(ret(1, 0) = **sin**(angle));

**return** ret;

}

## 3D Transform coordinate system Matrix

matrix **transformSystem**(**const** point3D& u, **const** point3D& v, **const** point3D& w) {

matrix ret = Identity();

**for** (**int** j = 0; j < 3; ++j) {

ret(0, j) = u[j];

ret(1, j) = v[j];

ret(2, j) = w[j];

}

**return** ret;

}

## 3D Inverse Transform coordinate system Matrix

matrix **ItransformSystem**(**const** point3D& u, **const** point3D& v, **const** point3D& w) {

matrix ret = Identity();

**for** (**int** j = 0; j < 3; ++j) {

ret(j, 0) = u[j];

ret(j, 1) = v[j];

ret(j, 2) = w[j];

}

**return** ret;

}

## 3D Get Perpendicular on two Vectors

**void** **getPrep**(point3D & w, point3D & v, point3D & u) {

w = w.Normalize();

**for** (**int** i = 0; i < 3; ++i) {

**if** (**fabs**(w[i]) > EPS) {

**int** j = (i + 1) % 3;

**int** k = (i + 2) % 3;

v[i] = w[j];

v[j] = -w[i];

v[k] = 0;

v = v.Normalize();

**break**;

}

}

u = v.Cross(w);

}

## 3D Rotation around General line Matrix

matrix **rotate**(**const** point3D& p, **const** point3D& q, **double** angle) {

point3D w((p - q).Normalize()), u, v;

getPrep(w, v, u);

**return** translate(p, 1) \* ItransformSystem(u, v, w) \* rotateZ(angle)

\* transformSystem(u, v, w) \* translate(p, -1);

}

## Line Plane Intersection

**bool** **linePlaneIntersect**(**const** point3D& p, **const** point3D& q, **const** point3D& pp,

**const** point3D& N, point3D& ret) {

**double** d = (q - p).Dot(N);

**if** (**fabs**(d) < EPS)

**return** **false**;

**double** t = (pp - p).Dot(N) / d;

ret = p + (q - p) \* t;

**return** **true**;

}

## Calculate the intersection of a line (not line segment) and a sphere

/\* -There are potentially two points of intersection given by

p = p1 + mu1 (p2 - p1)

p = p1 + mu2 (p2 - p1)

-To apply this to two dimensions, that is, the intersection of a line and a circle

simply remove the z component from the above mathematics.\*/

//If mu isn't between 0 and 1 then the intersection point isn't between p1,p2

**bool** **intersectLineSphere**(point3D p1, point3D p2, point3D sc, **double** r,

**double**& mu1, **double**& mu2) {

**double** a, b, c;

**double** bb4ac;

point3D dp;

dp.x() = p2.x() - p1.x();

dp.y() = p2.y() - p1.y();

dp.z() = p2.z() - p1.z();

a = dp.x() \* dp.x() + dp.y() \* dp.y() + dp.z() \* dp.z();

b = 2

\* (dp.x() \* (p1.x() - sc.x()) + dp.y() \* (p1.y() - sc.y())

+ dp.z() \* (p1.z() - sc.z()));

c = sc.x() \* sc.x() + sc.y() \* sc.y() + sc.z() \* sc.z();

c += p1.x() \* p1.x() + p1.y() \* p1.y() + p1.z() \* p1.z();

c -= 2 \* (sc.x() \* p1.x() + sc.y() \* p1.y() + sc.z() \* p1.z());

c -= r \* r;

bb4ac = b \* b - 4 \* a \* c;

**if** (**fabs**(a) < EPS || bb4ac < 0) {

mu1 = 0;

mu2 = 0;

**return** **false**;

}

mu1 = (-b + **sqrt**(bb4ac)) / (2 \* a);

mu2 = (-b - **sqrt**(bb4ac)) / (2 \* a);

**return** **true**;

}

## Tetrahedron centroid

point3D **tetra\_center**(**const** point3D & a, **const** point3D & b, **const** point3D & c,

**const** point3D & d) {

**return** (a + b + c + d) / 4;

}

## Tetrahedron volume

**double** **tetra\_volume**(**const** point3D & a, **const** point3D & b, **const** point3D & c,

**const** point3D & d) {

**return** **fabs**((a - d).Dot((b - d).Cross(c - d))) / 6;

}

## Spherical To Cartesian Coordiantes

/\*

-Note that rho represents the distance from the origin,

-phi (aka latitude) is the angle (in radians) between the vector from the origin to the point represneted by this coordinate and the z-axis

theta (aka longitude) is the angle (in radians) from the positive xz-plane to the point

\*/

**struct** spherical {

**double** rho, phi, theta;

};

cartesian **spherical2cartesian**(spherical sp) {

cartesian cp;

cp.x = sp.rho \* **cos**(sp.phi) \* **cos**(sp.theta);

cp.y = sp.rho \* **cos**(sp.phi) \* **sin**(sp.theta);

cp.z = sp.rho \* **sin**(sp.phi);

**return** cp;

}

spherical **cartesian2spherical**(cartesian cp) {

spherical sp;

sp.rho = sqrt(cp.x \* cp.x + cp.y \* cp.y + cp.z \* cp.z);

sp.phi = asin(cp.y / sp.rho);

sp.theta = cp.x >= 0 ? acos(cp.z / (sp.rho \* **cos**(sp.phi))) : -acos(

cp.z / (sp.rho \* **cos**(sp.phi)));

**return** sp;

}

# Math

## Numerical Integration

### Simpsons

**template**<**class** **T**>

**long** **double** **simpson**(**long** **double**(\*f)(**T** data, **const** **long** **double**&x), **T**& d,

**long** **double** a, **long** **double** b, **int** n = 100) {

**long** **double** h = (b - a) / n;

**long** **double** h2 = 0.5 \* h;

**long** **double** bound = a + (n - 0.25) \* h;

**long** **double** integral = (\*f)(d, a) + 4.0 \* (\*f)(d, a + h2);

**for** (a += h; a < bound; a += h)

integral += 2.0 \* (\*f)(d, a) + 4.0 \* (\*f)(d, a + h2);

**return** h \* (integral + (\*f)(d, a)) / 6;

}

### Adaptive Simpsons

//Adaptive Simpson works if there is no horizontal lines in the curve

inline double adaptiveSimpsonsAux(double (\*f)(double), double a,

double b, double epsilon, double S, double fa, double fb, double fc, int bottom) {

double c = (a + b) / 2, h = b - a;

double d = (a + c) / 2, e = (c + b) / 2;

double fd = f(d), fe = f(e);

double Sleft = (h / 12) \* (fa + 4 \* fd + fc);

double Sright = (h / 12) \* (fc + 4 \* fe + fb);

double S2 = Sleft + Sright;

if (bottom <= 0 || fabs(S2 - S) <= 15 \* epsilon)

return S2 + (S2 - S) / 15;

return adaptiveSimpsonsAux(f, a, c, epsilon / 2, Sleft, fa, fc, fd,

bottom - 1)

+ adaptiveSimpsonsAux(f, c, b, epsilon / 2, Sright, fc, fb, fe,

bottom - 1);

}

inline double adaptiveSimpsons(double (\*f)(double), // ptr to function

double a, double b, // interval [a,b]

double epsilon, // error tolerance

int maxRecursionDepth) { // recursion cap

double c = (a + b) / 2, h = b - a;

double fa = f(a), fb = f(b), fc = f(c);

double S = (h / 6) \* (fa + 4 \* fc + fb);

return adaptiveSimpsonsAux(f, a, b, epsilon, S, fa, fb, fc, maxRecursionDepth);

}

## Simplex

/\*

Simplex algorithm for solving linear programming problems.

O(N^3), where N is the number of variables

Testing Field: TopCoder(PreciousStones,Mixture), UVA(10498)

References:

-http://en.wikibooks.org/wiki/Operations\_Research/The\_Simplex\_Method

\*/

**#include**<cmath>

**#include**<vector>

**using** **namespace** std;

**enum** Type {

*LE*, *GE*, *EQ*

}; //respectively, less than or equal, greater than or equal, equal.

**enum** Result {

*OK*, *UNBOUND*, *UNFEASIBLE*

};

**enum** OFType {

*MAX*, *MIN*

}; //objective funtion type (maximize or minimize)

**#define** INF 1e30

**#define** EPS 1e-9

**#define** LD **long** **double** //Percision does matter in this algorithm

**struct** SimplexModel {

/\*\*\*\*\*Data Structures\*\*\*\*\*/

//Constraints

vector<vector<LD> > lhs; //matrix of constraints coefficients

vector<LD> rhs; //right hand side of constraints

vector<Type> constraintTypes; //type of constraint (greater than or equal, equal ... etc)

//Objective Function

vector<LD> of; //coefficients of variables in objective function

OFType oftype;

//Variables

vector<**bool**> unRestricted; //unRestricted[i] is true iff variable[i] can be -ve

//Values of variable in the solution (output only, don't fill)

vector<LD> solution;

//Internal use data structures (don't fill from outside)

**int** nVar, nCon; //number of variables/constraints

vector<**int**> negativePart; //index of negative part of unrestricted variables

vector<**int**> positivePart; //index of positive part of unrestricted variables

vector<**bool**> isNegativePart; //isNegativePart[i] = true iff variable i is x2 in (x=x1-x2)

vector<**int**> basic; //indicies of variables in the current solution (initially slacks and artificials)

vector<**bool**> isArtificial; //isArtificial[i] = true iff variable[i] is artificial

/\*\*\*\*\*Data Structures\*\*\*\*\*/

/\*\*\*\*\*\*Methods\*\*\*\*\*\*/

//Add new variable to the model (used to add slacks, artificials, negative parts and surpluses) and return its index

**int** **addVariable**() {

//Add variable to LHS

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

lhs[i].push\_back(0);

//Add varaible to Objective funtion

of.push\_back(0);

isArtificial.push\_back(**false**); //default value, might be modified later

isNegativePart.push\_back(**false**); //default value, might be modified later

positivePart.push\_back(0);

//Return variable index

**return** nVar++;

}

//Standardize model

**void** **standardize**() {

//Initialize internal data structures

nVar = unRestricted.size();

nCon = lhs.size();

negativePart.resize(nVar);

positivePart.resize(nVar);

isNegativePart.clear();

isNegativePart.resize(nVar, **false**);

basic.clear();

solution.clear();

solution.resize(nVar, 0);

isArtificial.clear();

isArtificial.resize(nVar, **false**);

**int** i, j, varIdx;

//Objective function should be max

**if** (oftype == *MIN*) {

**for** (i = 0; i < nVar; i++)

of[i] \*= -1;

oftype = *MAX*;

}

//Handle unresitricted variables (set x to x1-x2)

**for** (i = 0; i < unRestricted.size(); i++) {

**if** (!unRestricted[i])

**continue**;

varIdx = addVariable();

**for** (j = 0; j < nCon; j++)

lhs[j][varIdx] = -lhs[j][i];

of[varIdx] = -of[i];

negativePart[i] = varIdx;

positivePart[varIdx] = i;

isNegativePart[varIdx] = **true**;

}

//Standardize contstraints

**for** (i = 0; i < nCon; i++) {

**if** (rhs[i] < 0) {

rhs[i] \*= -1;

**for** (j = 0; j < nVar; j++)

lhs[i][j] \*= -1;

**if** (constraintTypes[i] != *EQ*)

constraintTypes[i] = constraintTypes[i] == *GE* ? *LE* : *GE*; //modify GE to LE and vice versa

}

//Add basic variable (variable in the initial solution, that is slack or artifical)

**int** basicVar = addVariable();

basic.push\_back(basicVar);

lhs[i][basicVar] = 1;

**switch** (constraintTypes[i]) {

**case** *GE*:

varIdx = addVariable(); //add surplus

lhs[i][varIdx] = -1;

**case** *EQ*:

isArtificial[basicVar] = **true**;

of[basicVar] = -INF;

**break**;

}

constraintTypes[i] = *EQ*;

}

}

//Solve model using Simplex algorithm

Result **solve**() {

//Standardize

standardize();

//Solve

**int** i, j, k;

LD z, ratio, cmz;

**while** (**true**) {

//Compute z, c-z and Select pivot column

**int** pivotCol = 0;

LD bestCMZ = -INF;

**for** (j = 0; j < nVar; j++) {

z = k = 0;

**for** (i = 0; i < basic.size(); i++)

z += of[basic[i]] \* lhs[k++][j];

cmz = of[j] - z;

pivotCol = (cmz > bestCMZ) ? j : pivotCol;

bestCMZ = max(cmz, bestCMZ);

}

//Check if no more improvement

**if** (fabs(bestCMZ) < EPS)

**break**;

//Compute ratio and Select pivot row

**int** pivotRow = 0;

LD bestRatio = INF;

**for** (i = 0; i < nCon; i++) {

**if** (lhs[i][pivotCol] < EPS)

**continue**; //avoid division by zero

ratio = rhs[i] / lhs[i][pivotCol];

**if** (ratio < 0)

ratio = INF; //to avoid selecting negative ratios

pivotRow = ratio < bestRatio ? i : pivotRow;

bestRatio = min(bestRatio, ratio);

}

**if** (bestRatio >= INF)

**return** *UNBOUND*; //unbounded solution (can achieve infinite profit)

//Update table

basic[pivotRow] = pivotCol;

//Set coeff of new basic to 1

LD pivot = lhs[pivotRow][pivotCol];

**for** (i = 0; i < nVar; i++)

lhs[pivotRow][i] /= pivot;

rhs[pivotRow] /= pivot;

//Set coeff of pivotCol to 0

**for** (i = 0; i < nCon; i++) {

**if** (i == pivotRow)

**continue**;

LD val = -lhs[i][pivotCol];

**for** (j = 0; j < nVar; j++)

lhs[i][j] += lhs[pivotRow][j] \* val;

rhs[i] += rhs[pivotRow] \* val;

}

}

//Compute solution

**for** (i = 0; i < basic.size(); i++) {

**if** (isArtificial[basic[i]] && fabs(rhs[i]) > EPS)

**return** *UNFEASIBLE*;

**if** (basic[i] < solution.size())

solution[basic[i]] += rhs[i];

**else** **if** (isNegativePart[basic[i]])

solution[positivePart[basic[i]]] += -rhs[i];

}

**return** *OK*;

}

/\*\*\*\*\*\*Methods\*\*\*\*\*\*/

};

////////////////

**#include**<numeric>

**class** PreciousStones {

**public**:

LD **value**(vector<**int**> silver, vector<**int**> gold) {

**int** i, j, N = silver.size();

**int** nCon = N + 1;

**int** nVar = N;

SimplexModel model;

//Objective funtion

**for** (i = 0; i < silver.size(); i++)

model.of.push\_back(silver[i]);

model.oftype = *MAX*;

//Constraints

model.unRestricted.resize(nVar, **false**);

model.constraintTypes.resize(nCon, *LE*);

model.lhs.resize(nCon, vector<LD> (nVar, 0));

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

model.rhs.push\_back(1);

model.lhs[i][i] = 1;

}

model.rhs.push\_back(accumulate(gold.begin(), gold.end(), 0));

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

model.lhs.back()[i] = silver[i] + gold[i];

Result r = model.solve();

LD d = 0;

**for** (i = 0; i < model.solution.size(); i++)

d += model.solution[i] \* silver[i];

**return** d;

}

};

# Other

## Closest Pair of Points O(N lg N)

**#define** type **double**

**#define** MapIterator map<type, multiset<type> >::iterator

**#define** SetIterator multiset<type>::iterator

**const** **int** SIZE = 10000; //Maximum number of points

type x[SIZE], y[SIZE]; //Coordinates of points

**int** N; //Number of points

**double** INF = INT\_MAX;

**double** **getClosestPair**() {

map<type, multiset<type> > points;

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

points[x[i]].insert(y[i]);

**double** d = INF;

**for** (MapIterator xitr1 = points.begin(); xitr1 != points.end(); xitr1++){

**for** (SetIterator yitr1 = (\*xitr1).second.begin(); yitr1!= (\*xitr1).second.end(); yitr1++) {

type x1 = (\*xitr1).first, y1 = \*yitr1;

MapIterator xitr3 = points.upper\_bound(x1 + d);

**for** (MapIterator xitr2 = xitr1; xitr2 != xitr3; xitr2++)

{

type x2 = (\*xitr2).first;

SetIterator yitr2 = (\*xitr2).second.lower\_bound(y1 - d);

SetIterator yitr3 = (\*xitr2).second.upper\_bound(y1 + d);

**for** (SetIterator yitr4 = yitr2; yitr4 != yitr3; yitr4++) {

**if** (xitr1 == xitr2 && yitr1 == yitr4)

**continue**; //same point type y2 = \*yitr4;

d = min(d, hypot(x1 - x2, y1 - y2));

}

}

}

}

**return** d;

}

## CCW

**typedef** complex<**double**> P;

**namespace** std {

**bool** **operator <**(**const** P& a, **const** P& b) {

**return** real(a) != real(b) ? real(a) < real(b) : imag(a) < imag(b);

}

}

**int** **ccw**(P a, P b, P c) {

b -= a;

c -= a;

**if** (cross(b, c) > 0)

**return** +1; // counter clockwise

**if** (cross(b, c) < 0)

**return** -1; // clockwise

**if** (dot(b, c) < 0)

**return** +2; // c--a--b on line

**if** (norm(b) < norm(c))

**return** -2; // a--b--c on line

**return** 0;

}

## Max Empty Rectangle

// Given cells

// Doesn’t consider obstacles on boundaries (boundaries are empty cells)

**const** **int** MAX = 3000;

**class** MaxEmptyRect {

**private**:

**int** W, H, N;

vector<**int**> dCols[MAX + 2];

**int** pLeft[MAX + 1], pRight[MAX + 1], pTop[MAX + 1];

**int** best;

**public**:

**MaxEmptyRect**(vector<pair<**int**, **int**> > points, **int** height, **int** width) {

H = height;

W = width;

N = points.size();

best = 0;

**memset**(pLeft, 0, **sizeof**(pLeft));

**memset**(pRight, 0, **sizeof**(pRight));

**memset**(pTop, 0, **sizeof**(pTop));

**int** i;

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

**int** r = points[i].first, c = points[i].second;

dCols[r].push\_back(c);

}

**for** (i = 0; i <= H; i++) {

dCols[i].push\_back(0);

dCols[i].push\_back(W + 1);

sort(dCols[i].begin(), dCols[i].end());

}

**int** k;

**for** (i = 1; i <= H; i++) {

k = 0;

**for** (**int** j = 1; j <= W; j++) {

**if** (dCols[i][k + 1] == j) {

k++;

pTop[j] = i;

**continue**;

}

**if** (pTop[j] + 1 == i) {

pLeft[j] = dCols[i][k];

pRight[j] = dCols[i][k + 1];

} **else** {

pLeft[j] = dCols[i][k] > pLeft[j] ? dCols[i][k] : pLeft[j];

pRight[j] = dCols[i][k + 1] < pRight[j] ? dCols[i][k + 1]

: pRight[j];

}

**int** area = (i - pTop[j]) \* (pRight[j] - pLeft[j] - 1);

best = area > best ? area : best;

}

}

}

**int** **getMaxEmptyArea**() {

**return** best;

}

};

//MAIN

MaxEmptyRect m(vec, l, w);

cout << m.getMaxEmptyArea() << endl;

// Buggy (msh sha8al [[fakss]])

Max empty rectangle, On border, O(N^2)

**#define** point pair<**int**,**int**>

**class** MaxEmptyRect {

**private**:

vector<point> P;

**int** l,w;

**int** best;

**void** update(**int** a) {best = a > best ? a : best;}

**void** split(**int** i, **int** y0, **int** y1) {

**if**(l\*(y1-y0) < best)**return**;

**int** px,py;

**if** (y0==y1) **return**;

**if** (i==P.size())update(l\*(y1-y0));

**else** {

px=P[i].first;

py=P[i].second;

**if** (y0<=py && py<=y1) {

update( px\*(y1-y0) );

split(i+1,y0,py);

split(i+1,py,y1);

} **else** split(i+1,y0,y1);

}

}

**void** sweep() {

**int** i,j, y0,y1, pix,piy,pjx,pjy;

**for** (i=0; i<P.size(); i++) {

y0=0; y1=w;

pix=P[i].first; piy=P[i].second;

**for** (j=i+1; j<P.size(); j++) {

pjx=P[j].first; pjy=P[j].second;

**if** (y0<=pjy && pjy <=y1) {

update( (pjx-pix)\*(y1-y0) );

**if** (pjy<piy)y0=pjy;

**else** **if** (pjy>piy)y1=pjy;

**else** **break**;

}

}

**if** (j==P.size())update( (l-pix)\*(y1-y0) );

}

}

**public**:

MaxEmptyRect(vector<pair<**int**, **int**> > points, **int** height, **int** width) {

P = points;

l = height;

w = width;

best = 0;

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

split(0,0,w);

sweep();

}

**int** getArea() {**return** best;}

};

//MAIN

MaxEmptyRect m(vec, l, w);

cout << m.getArea() << endl;

## LIS O(N lg K)

#define MAX 100005

#define oo 1e9

int arr[MAX], len[MAX], par[MAX], n;

bool cmp(int a, int b) {

return arr[a] < arr[b];

}

int LIS() {

arr[n] = -oo;

len[0] = n;

int res = 0;

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

int idx = lower\_bound(len, len + res + 1, i, cmp) - len;

res = max(res, idx);

len[idx] = i;

par[i] = len[idx - 1];

}

return res;

}

int LISres[MAX];

void buildSequence(int lastIdx, int pos) {

for (; pos >= 0; pos--) {

LISres[pos] = arr[lastIdx];

lastIdx = par[lastIdx];

}

}

int main() {

ios\_base::sync\_with\_stdio(0);

cin.tie(0);

cin >> n;

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

cin >> arr[i];

int res = LIS();

buildSequence(len[res], res - 1);

cout << res << endl;

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

cout << LISres[i] << " ";

cout << endl;

return 0;

}

## RMQ

const int MAXN = 100002;

typedef long long ll;

int Log[MAXN];

ll stable[MAXN][17]; // sparse table

ll arr[MAXN];

int n;

void build(){

int cnt = -1;

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

if(!((i+1)&i)) cnt++;

stable[i][0] = i;

Log[i+1] = cnt;

}

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

for(int i = 0; (i+(1<<j)) <= n; i++){

int a = stable[i][j-1];

int b = stable[i + (1<<(j-1))][j-1];

stable[i][j] = ((arr[a]<arr[b])?a:b);

}

}

}

int getMin(int st, int en){

int L = Log[en-st+1];

int a = stable[st][L], b = stable[en-(1<<L)+1][L];

return ((arr[a]<arr[b])?a:b);

}

## LCA

// don't forget to update size

const int siz = 10001;

int n;

// you need to calculate lvl and anc

int lvl[siz], anc[siz][25];

void buildLCA() {

// lvl contains the level of each node 0-based

// for each node i, anc[i][0] = parent of node i

int lg = ceil(log2(n));

FOR (j , 1 , lg)

FOR (i , 0 , n)

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

}

int LCA(int i, int j) { // returns node ID (LCA for i, j)

int lg = ceil(log2(n));

int st = lg;

if (lvl[i] > lvl[j])

swap(i, j);

int cur = lvl[j];

for (; st >= 0; st--)

if (cur - (1 << st) >= lvl[i])

cur -= (1 << st), j = anc[j][st];

if (i == j)

return 2 \* i - j;

cur = lvl[i];

for (st = lg; st >= 0; st--)

if (anc[i][st] != anc[j][st])

cur -= (1 << st), i = anc[i][st], j = anc[j][st];

return anc[i][0];

}

## LCA on DAG

**#define** MX 1000

**typedef** vector<vector<**int**> > vii;

**#define** pb push\_back

**int** in1[MX];

**int** in2[MX];

**int** out[MX];

vii g, gr;

**int** n, m;

bitset<MX> des[MX];

bitset<MX> anc[MX];

**int** ind[MX];

bitset<MX> vis[MX];

**int** mat[MX][MX];

**void** **calc1**() {

mem(des, 0);

queue<**int**> q;

rep(i,n)

**if** (!out[i])

q.push(i);

**while** (!q.empty()) {

**int** t = q.front();

q.pop();

des[t][t] = 1;

rep(i,sz(g[t]))

des[t] |= des[g[t][i]];

rep(i,sz(r[t])) {

out[gr[t][i]]--;

**if** (!out[gr[t][i]])

q.push(gr[t][i]);

}

}

}

**void** **calc2**() {

mem(anc, 0);

queue<**int**> q;

rep(i,n)

**if** (!in1[i])

q.push(i);

**int** cur = 0;

**while** (!q.empty()) {

**int** t = q.front();

ind[t] = cur++;

q.pop();

anc[t][t] = 1;

rep(i,sz(gr[t]))

anc[t] |= anc[gr[t][i]];

rep(i,sz(g[t])) {

in1[g[t][i]]--;

**if** (!in1[g[t][i]])

q.push(g[t][i]);

}

}

}

**void** **calc3**() {

mem(anc, 0);

queue<**int**> q;

rep(i,n)

**if** (!in2[i]) {

q.push(i);

rep(j,n)

**if** (des[i][j]) {

mat[i][j] = mat[j][i] = i;

vis[i][j] = vis[j][i] = 1;

}

}

**while** (!q.empty()) {

**int** t = q.front();

q.pop();

rep(i,sz(gr[t])) {

rep(j,n) {

**if** (des[t][j])

mat[t][j] = mat[j][t] = t;

**else** **if** (des[j][t])

mat[t][j] = mat[j][t] = j;

**else** {

**if** (!vis[j][t] || ind[mat[gr[t][i]][j]] > ind[mat[j][t]])

mat[t][j] = mat[j][t] = mat[gr[t][i]][j];

}

vis[t][j] = vis[j][t] = 1;

}

}

rep(i,sz(g[t])) {

in2[g[t][i]]--;

**if** (!in2[g[t][i]])

q.push(g[t][i]);

}

}

}

**void** **init**() {

g = vector<vector<**int**> >(n);

gr = vector<vector<**int**> >(n);

mem(in1, 0);

mem(in2, 0);

mem(out, 0);

}

**void** **addEdge**(**int** from, **int** to) {

g[from].pb(to);

gr[to].pb(from);

in1[to]++;

in2[to]++;

out[from]++;

}

**void** **calc**() {

calc1();

calc2();

mem(vis, 0);

calc3();

}

## BIT

**//insert 5 3 9, put 1 at 5, 3 and 9. add(5, 1), add(3, 1), add(9, 1);**

**//find(3) returns 9, find(2) returns 5, find(1) returns 3 //find is lower bound**

**//get(9) returns 3, get(5) returns 2, get(3) returns 1**

**struct** BIT {

vector<**long** **long**> v;

**BIT**(**int** s) {

resize(s);

}

**void** **clear**() {

v.clear();

}

**BIT**() {

}

**void** **resize**(**int** s) {

s = 1 << (**int**) ceil(**log**(1.0 \* s) / **log**(2.) + EPSILON);

v.resize(s);

}

**long** **long** **get**(**int** i) {

i++;

**long** **long** r = 0;

**while** (i) {

r += v[i - 1];

i -= i & -i;

}

**return** r;

}

**void** **add**(**int** i, **long** **long** val) {

i++;

**while** (i <= (**int**) v.size()) {

v[i - 1] += val;

i += i & -i;

}

}

**int** **find**(**long** **long** val) {

**int** s = 0;

**int** m = v.size() >> 1;

**while** (m) {

**if** (v[s + m - 1] < val)

val -= v[(s += m) - 1];

m >>= 1;

}

**return** s;

}

};

## BIT Update Range

**const** **int** siz = (1 << 21);

**long** **long** a[siz], b[siz];

**void** **add**(**int** i, **long** **long** valA, **long** **long** valB) {

i++;

**while** (i <= siz) {

a[i - 1] += valA;

b[i - 1] += valB;

i += i & -i;

}

}

**long** **long** **get**(**int** i) {

**int** ii = i;

i++;

**long** **long** res = 0;

**while** (i) {

res += a[i - 1] + b[i - 1] \* ii;

i -= i & -i;

}

**return** res;

}

**void** **addRange**(**int** st, **int** en, **long** **long** val) {

**int** si = en - st + 1;

add(st, -val \* (st - 1), val);

// add(en + 1, val \* (st - 1) + val \* si, -val);

add(en + 1, val \* en, -val);

}

## 2D BIT

**int** arr[R][C], mat[R][C];

**void** **add**(**int** i, **int** jj, **int** v) {

i++;

jj++;

**while** (i <= R) {

**int** j = jj;

**while** (j <= C) {

arr[i - 1][j - 1] += v;

j += (j & -j);

}

i += (i & -i);

}

}

**int** **get**(**int** i, **int** jj) {

**int** v = 0;

i++;

jj++;

**while** (i) {

**int** j = jj;

**while** (j) {

v += arr[i - 1][j - 1];

j -= (j & -j);

}

i -= (i & -i);

}

**return** v;

}

**int** **get2D**(**int** b, **int** l, **int** t, **int** r) {

**int** v = 0;

v += get(t, r);

v -= get(t, l - 1);

v -= get(b - 1, r);

v += get(b - 1, l - 1);

**return** v;

}

## Suffix Arrays (Old versions)

**#include**<iostream>

**#include**<cstdio>

**using** **namespace** std;

**#define** Max\_N 1000

### O(N^2 lg N)

// buildSA O(n^2 log(n) )

**char** str[Max\_N];

**int** suffix[Max\_N];

**struct** comp {

**bool** **operator()**(**int** a, **int** b) **const** {

**return** **strcmp**(str + a, str + b) < 0;

}

};

**void** **buildSA**() {

**int** n;

**for** (n = 0; n == 0 || str[n - 1]; n++)

suffix[n] = n;

sort(suffix, suffix + n, comp());

}

### O(N (lg N)^2)

// buildSA O(n(logn)^2)

**char** str[Max\_N];

**int** suffix[Max\_N];

**int** group[Max\_N];

**int** tg[Max\_N];

**struct** comp {

**int** h;

**comp**(**int** h) :

h(h) {

}

**bool** **operator ()**(**const** **int**& s1, **const** **int**& s2) **const** {

**return** group[s1] < group[s2] || group[s1] == group[s2] && group[s1 + h]

< group[s2 + h];

}

};

**void** **buildSA**() {

**int** n;

**for** (n = 0; n == 0 || str[n - 1]; n++) {

suffix[n] = n;

group[n] = str[n];

}

sort(suffix, suffix + n, comp(0));

tg[0] = tg[n - 1] = 0;

**for** (**int** h = 1; tg[n - 1] != n - 1; h <<= 1) {

comp c(h);

sort(suffix, suffix + n, c);

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

tg[i] = tg[i - 1] + c(suffix[i - 1], suffix[i]);

}

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

group[suffix[i]] = tg[i];

}

}

}

### O(N lg N)

// buildSA O(nlogn)

**char** str[Max\_N];

**int** suffix[Max\_N];

**int** group[Max\_N];

**int** tg[Max\_N < 128 ? 128 : Max\_N];

**int** newSuffix[Max\_N];

**int** gstart[Max\_N];

**void** **buildSA**() {

**int** n;

**memset**(tg, -1, (**sizeof** tg[0]) \* 128);

**for** (n = 0; n == 0 || str[n - 1]; n++) {

newSuffix[n] = tg[str[n]];

tg[str[n]] = n;

}

**int** ng = -1, j = 0;

**for** (**int** i = 0; i < 128; ++i) {

**if** (tg[i] != -1) {

gstart[++ng] = j;

**int** cur = tg[i];

**while** (cur != -1) {

suffix[j++] = cur;

group[cur] = ng;

cur = newSuffix[cur];

}

}

}

tg[0] = tg[n - 1] = 0;

newSuffix[0] = suffix[0];

**for** (**int** h = 1; tg[n - 1] != n - 1; h <<= 1) {

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

j = suffix[i] - h;

**if** (j < 0)

**continue**;

newSuffix[gstart[group[j]]++] = j;

}

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

**bool** newgroup = group[newSuffix[i - 1]] < group[newSuffix[i]]

|| group[newSuffix[i - 1]] == group[newSuffix[i]]

&& group[newSuffix[i - 1] + h] < group[newSuffix[i]

+ h];

tg[i] = tg[i - 1] + newgroup;

**if** (newgroup)

gstart[tg[i]] = i;

}

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

suffix[i] = newSuffix[i];

group[suffix[i]] = tg[i];

}

}

}

### LCP

**int** rank[Max\_N];

**int** lcp[Max\_N];

**void** **buildLCP**() {

**int** n;

**for** (n = 0; n == 0 || str[n - 1]; n++)

rank[suffix[n]] = n;

**int** c = 0;

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

**if** (rank[i]) {

**int** j = suffix[rank[i] - 1];

**while** (str[i + c] == str[j + c])

c++;

}

lcp[rank[i]] = c;

**if** (c)

c--;

}

}

struct cmp {

int k;

cmp(int \_k) {

k = \_k;

}

bool operator ()(const int &i, const int &j)const {

return str[i+k]<str[j+k];

}

};

//if u search for small strings in a large string use suffix array with this method to search for these small strings using binary search

bool search(char \*cur) {

int s = 0, e = strlen(str) + 1;

int f = 1;

for (int j = 0; cur[j]; ++j) {

s = lower\_bound(suffix+s, suffix+e,

cur-str, cmp(j)) - suffix;

e = upper\_bound(suffix+s, suffix+e, cur-str,

cmp(j)) - suffix;

if(s >= e) {

f = 0;

break;

}

}

return f;

}

## Suffix Arrays (NlogN)

const int siz = 200005;

char s[siz];

// idx -> suffix position in the sorted order according to the current prefix length

// val -> start position of suffix inside the string

int suff[siz];

// idx -> start position of suffix inside the string

// val -> suffix order in the list of sorted suffixes according to the current prefix length

int order[siz];

// idx -> position of suffix in the current "suff" array

// val -> suffix order in the list of sorted suffixes according to the current prefix length

int newOrder[siz];

// idx -> value from "order"

// val -> idx in "suff"

int groupStart[siz];

// copy of "suff" but sorted 3la 2 \* len

int newSuff[siz];

// meen el suffixes elli btebda2 bel 7arf da

int head[128], nxt[siz];

struct cmp {

int len;

cmp(int len) :

len(len) { // Initialization list

}

bool operator ()(const int &a, const int &b) const {

return order[a] < order[b]

|| (order[a] == order[b] && order[a + len] < order[b + len]);

}

};

void print(int \*arr = { 0 }) {

for (int i = 0; !i || s[i - 1]; i++)

cout << (char\*) (s + newSuff[i]) << endl;

cout << endl;

}

void suffixArrays() {

mem(head, -1);

int len = 0;

for (; !len || s[len - 1]; len++) {

nxt[len] = head[s[len]];

head[s[len]] = len;

}

int ng = -1;

for (int i = 0, j = 0; i < 128; i++) {

int cur = head[i];

// combo loop

for (cur != -1 && (groupStart[++ng] = j); cur != -1; cur = nxt[cur]) {

suff[j++] = cur;

order[cur] = ng;

}

}

newSuff[0] = suff[0];

newOrder[len - 1] = -1;

for (int cur = 1; newOrder[len - 1] != len - 1; cur <<= 1) {

cmp c(cur);

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

int j = suff[i] - cur;

if (j < 0)

continue;

newSuff[groupStart[order[j]]++] = j;

}

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

bool ngroup = c(newSuff[i - 1], newSuff[i]);

newOrder[i] = newOrder[i - 1] + ngroup;

if (ngroup)

groupStart[newOrder[i]] = i;

}

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

suff[i] = newSuff[i], order[suff[i]] = newOrder[i];

}

}

int lcp[siz];

void buildLCP() {

int cnt = 0;

for (int i = 0; s[i]; i++) {

int j = suff[order[i] - 1];

while (s[i + cnt] == s[j + cnt])

cnt++;

lcp[order[i]] = cnt;

if (cnt)

cnt--;

}

}

## Suffix Tree

**struct** edge {

**int** to, s, e;

**edge**(**int** to, **int** s, **int** e) :

to(to), s(s), e(e) {

}

**edge**() {

}

};

**struct** \_hash {

**int** **operator ()**(**const** pair<**int**, **char**>& t) **const** {

**return** t.first \* 257 + t.second;

}

};

**char** str[MAXSIZE];

**int** size, strNum, mx = 0, nnodes;

hash\_map<pair<**int**, **char**> , edge,\_hash> g;

**typedef** hash\_map<pair<**int**, **char**> , edge,\_hash>::iterator iter;

vector<**int**> res, f;

**bool** **getEdge**(**int** s, **char** t, **int**& kd, **int**&pd, **int**&sd) {

**if** (s == -1) {

sd = kd = pd = 0;

**return** **true**;

}

iter it = g.find(make\_pair(s, t));

**if** (it == g.end())

**return** **false**;

kd = it->second.s;

pd = it->second.e;

sd = it->second.to;

**return** **true**;

}

pair<**int**, **int**> **canonize**(**int** s, **int** k, **int** p) {

**if** (p < k)

**return** make\_pair(s, k);

**int** kd, pd, sd;

getEdge(s, str[k], kd, pd, sd);

**while** (pd - kd <= p - k) {

k += pd - kd + 1;

s = sd;

**if** (k <= p)

getEdge(s, str[k], kd, pd, sd);

}

**return** make\_pair(s, k);

}

**void** **init**() {

g.clear();

f.clear();

g.resize(size \* 2);

f.reserve(size \* 2);

nnodes = 1;

f.push\_back(-1);

}

pair<**bool**, **int**> **test\_and\_split**(**int** s, **int** k, **int** p, **char** t) {

**int** kd, pd, sd;

**if** (k <= p) {

getEdge(s, str[k], kd, pd, sd);

**if** (t == str[kd + p - k + 1])

**return** make\_pair(**true**, s);

**int** r = nnodes++;

f.push\_back(-1);

g[make\_pair(s, str[kd])] = edge(r, kd, kd + p - k);

g[make\_pair(r, str[kd + p - k + 1])] = edge(sd, kd + p - k + 1, pd);

**return** make\_pair(**false**, r);

}

**return** make\_pair(getEdge(s, t, kd, pd, sd), s);

}

pair<**int**, **int**> **update**(**int** s, **int** k, **int** i) {

**int** oldr = 0;

pair<**bool**, **int**> temp = test\_and\_split(s, k, i - 1, str[i]);

**while** (!temp.first) {

**int** r = temp.second;

**int** rd = nnodes++;

f.push\_back(-1);

g[make\_pair(r, str[i])] = edge(rd, i, size);

**if** (oldr)

f[oldr] = r;

oldr = r;

pair<**int**, **int**> c = canonize(f[s], k, i - 1);

s = c.first;

k = c.second;

temp = test\_and\_split(s, k, i - 1, str[i]);

}

**if** (oldr)

f[oldr] = s;

**return** make\_pair(s, k);

}

**void** **insert**() {

size = **strlen**(str) - 1;

pair<**int**, **int**> temp(0, 0); // s,k

**int** i = 0;

init();

**while** (str[i]) {

temp = update(temp.first, temp.second, i);

temp = canonize(temp.first, temp.second, i++);

}

}

vector<vector<**char**> > adj;

vector<pair<**int**, **char**> > parent;

**void** **constructAdjacency**() {

adj.clear();

adj.resize(f.size());

parent.clear();

parent.resize(f.size());

parent[0] = make\_pair(-1, -1);

iter it = g.begin();

**for** (; it != g.end(); it++) {

adj[it->first.first].push\_back(str[it->second.s]);

parent[it->second.to] = make\_pair(it->first.first, str[it->second.s]);

}

}

**void** **sortAdjacency**() {

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

sort(adj[i].begin(), adj[i].end());

}

**int** n, m, s2;

vector<**int**> bestNode;

**int** len[100], strInd[MAXSIZE];

vector<pair<**int**, **int**> > que;

**void** **bfs**() {

**int** i, sz;

que.clear();

que.push\_back(make\_pair(0, 0));

**for** (**int** ind = 0; ind < que.size(); ind++)

**for** (i = 0; i < adj[que[ind].first].size(); i++) {

iter it = g.find(make\_pair(que[ind].first, adj[que[ind].first][i]));

que.push\_back(

make\_pair(it->second.to,

que[ind].second + it->second.e - it->second.s + 1));

}

}

**void** **findLongest**() {

**int** best = -1;

vector<bitset<100> > has(f.size());

iter it; // empty string is not counted as common substring, to count it, make i >= 0 in the for loop, and ensure that it != g.end()

**for** (**int** i = que.size() - 1; i > 0; i--) {

it = g.find(parent[que[i].first]);

**int** ind = strInd[it->second.s];

**if** (strInd[it->second.s] != strInd[it->second.e + 1])

has[que[i].first][ind] = 1;

**if** (i != 0)

has[parent[que[i].first].first] |= has[que[i].first];

**if** (has[que[i].first].count() > strNum / 2) {

**if** (que[i].second > best)

bestNode.clear(), best = que[i].second;

**if** (que[i].second == best)

bestNode.push\_back(que[i].first);

}

}

}

**void** **printPrefix**(**int** node, string &s) {

**if** (node != 0) {

printPrefix(parent[node].first, s);

iter it = g.find(parent[node]);

**for** (**int** i = it->second.s; i <= it->second.e; i++)

s += str[i];

}

}

## KMP

// Memoize this function in case of TLE :P

int getLen(int len, const string& p, char c, const vi& f) {

while (len && c != p[len])

len = f[len - 1];

if (c == p[len])

len++;

return len;

}

vi computePrefix(const string &p) {

vi f(1, 0);

f.reserve(p.size());

int len = 0;

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

len = getLen(len, p, p[i], f);

f.push\_back(len);

}

return f;

}

vi findLocs(const string &s, const string &p) {

vi f = computePrefix(p), res;

int len = 0;

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

len = getLen(len, p, s[i], f);

if (len == p.size()) {

res.push\_back(i - len + 1);

len = f.back();

}

}

return res;

}

## Rabin Karp

### V1

//ll md[3] = {2000000063, 2000000087, 2000000089};

//ll mdi[3] = {622568113, 661478628, 1712062333};

//ll bs = 257; ll md=2147483629;

ll bs = 53;

ll mdi = 1053482535;

ll **pow**(ll n, ll p) {

**if** (p == 0)

**return** 1;

ll t = pow(n, p / 2) % md;

**if** (p % 2)

**return** (((t \* t) % md) \* n) % md;

**return** ((t \* t) % md);

}

ll **addDigit**(ll n, ll val, ll ind) {

ll temp = (pow(bs, ind) \* val) % md;

**return** (n + temp) % md;

}

ll **shiftLeft**(ll n) {

**return** (n \* bs) % md;

}

ll **shiftRight**(ll n) {

**return** (n \* mdi) % md;

}

ll **removeDigit**(ll n, ll val, ll ind) {

ll temp = (pow(bs, ind) \* val) % md;

**return** (n + md - temp) % md;

}

### V2

**#define** BASE 128LL

**#define** BASEINV 1453125008LL

**#define** MOD 2000000011LL

ll **addCharAt**(**int** ind, **char** v, ll pvHashV) {

**return** ((pow(BASE, (ll) ind) \* v) % MOD + pvHashV) % MOD;

}

ll **removeCharAt**(**int** ind, **char** v, ll pvHashV) {

**return** (MOD - (((pow(BASE, (ll) ind) \* v) % MOD)) % MOD + pvHashV) % MOD;

}

ll **shiftL**(ll pvHash) {

**return** (pvHash \* BASE) % MOD;

}

ll **shiftR**(ll pvHash) {

**return** (pvHash \* BASEINV) % MOD;

}

### V3

const int MOD = 1e9 + 9;

const int base = (srand(time(0)), 128 + rand() % 200);

struct MUL {

int operator()(const int &a, const int &b) const {

return a \* (long long) b % MOD;

}

};

int identity\_element(const MUL &m) {

return 1;

}

//const int inv = power(base, MOD - 2, MUL());

int main() {

#ifndef ONLINE\_JUDGE

freopen("test.in", "rt", stdin);

// freopen("o.txt", "wt", stdout);

#endif

MUL mul;

int k;

cin >> k;

int h1, h2;

h1 = h2 = 0;

string s;

cin >> s;

int p = 1;

for (int i = 0, j = k - 1; i < k; i++, j--) {

if (i)

p = mul(p, base);

h1 = mul(h1, base);

h2 = mul(h2, base);

h1 = (h1 + s[i]) % MOD;

h2 = (h2 + s[j]) % MOD;

}

int res = 0;

for (int i = 0, j = k; j <= s.size(); i++, j++) {

res += (h1 == h2);

h1 = (h1 - mul(s[i], p) + MOD) % MOD;

h1 = mul(h1, base);

h1 = (h1 + s[j]) % MOD;

h2 = (h2 - s[i] + MOD) % MOD;

h2 = mul(h2, inv);

h2 = (h2 + mul(s[j], p)) % MOD;

}

cout << res << "\n";

return 0;

}

## Aho

struct HASH {

int operator()(const pair<int, char>&p) const {

return p.first \* 128 + p.second;

}

};

unordered\_map<pair<int, char>, int, HASH> child;

vector<vector<char> > childChar;

vector<vector<int> > patIdx;

vector<int> fail;

int addNode() {

childChar.push\_back(vector<char>());

patIdx.push\_back(vector<int>());

fail.push\_back(-1);

return patIdx.size() - 1;

}

void init() {

child.clear();

childChar.clear();

patIdx.clear();

addNode();

}

void insert(const string &s, int idx) {

int curr = 0;

for (char c : s) {

int &nxt = child.insert(mp(mp(curr, c), -1)).first->second;

if (nxt == -1) {

nxt = addNode();

childChar[curr].push\_back(c);

}

curr = nxt;

}

patIdx[curr].push\_back(idx);

}

int nxtChar(int f, char c) {

while (!child.count(mp(f, c)))

f = fail[f];

f = child[mp(f, c)];

return f;

}

void buildFailure() {

queue<int> q;

for (int i = 0; i < 128; i++) {

int x = child.insert(mp(mp(0, i), 0)).first->second;

if (x) {

fail[x] = 0;

q.push(x);

}

}

while (q.size()) {

int cur = q.front();

q.pop();

for (char c : childChar[cur]) {

int chld = child[mp(cur, c)];

int f = fail[cur];

f = nxtChar(f, c);

fail[chld] = f;

patIdx[chld].insert(patIdx[chld].end(), patIdx[f].begin(),

patIdx[f].end());

q.push(chld);

}

}

}

void buildAho(vs &v) {

init();

FOR (i , 0 , sz(v))

insert(v[i], i);

buildFailure();

}

vector<bool> find(const string &s, int np) {

int cur = 0;

vector<bool> res(np);

for (char c : s) {

cur = nxtChar(cur, c);

FOR (i , 0 , sz(patIdx[cur]))

res[patIdx[cur][i]] = 1;

}

return res;

}

## 2 SAT

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

#define pb push\_back

#define sz(v) (int)v.size()

#define all(c) (c).begin(),(c).end()

#define mem(s,v) memset(s,v,sizeof(s))

typedef vector<int> vi;

const int MAX = 130, MAXE = 130 \* 130;

int n;

int head[MAX], nxt[MAXE], to[MAXE];

int edgeCount;

void init() {

edgeCount = 0;

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

}

void addEdge(int f, int t) {

nxt[edgeCount] = head[f];

head[f] = edgeCount;

to[edgeCount++] = t;

}

int low[MAX], tim[MAX], curtime;

int stk[MAX], stksz;

int compId[MAX], compnum;

void tarjanDFS(int cur) {

low[cur] = tim[cur] = curtime++;

stk[stksz++] = cur;

for (int i = head[cur]; i != -1; i = nxt[i]) {

int j = to[i];

if (compId[j] == -1) {

if (tim[j] == -1)

tarjanDFS(j);

low[cur] = min(low[cur], low[j]);

}

}

if (low[cur] == tim[cur]) {

do {

compId[stk[stksz - 1]] = compnum;

} while (stk[--stksz] != cur);

compnum++;

}

}

void SCC() {

compnum = 0;

curtime = 0;

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

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

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

if (tim[i] == -1)

tarjanDFS(i);

}

}

int nodeID(int cur) {

return 2 \* cur;

}

int NOT(int cur) {

return cur ^ 1;

}

void addOR(int i, int j) {

addEdge(NOT(i), j);

addEdge(NOT(j), i);

}

int invComp[MAX], sortedOrder[MAX], in[MAX], sorSize;

vector<vi> adjComp;

void topo() {

mem(in, 0), sorSize = 0;

FOR (i , 0 , sz(adjComp))

FOR (k , 0 , sz(adjComp[i]))

in[adjComp[i][k]]++;

queue<int> q;

FOR (i , 0 , sz(adjComp))

if (!in[i])

q.push(i);

while (sz(q)) {

int i = q.front();

q.pop();

sortedOrder[sorSize++] = i;

FOR (k , 0 , sz(adjComp[i]))

if (!--in[adjComp[i][k]])

q.push(adjComp[i][k]);

}

}

int compres[MAX];

bool \_2sat() {

SCC();

FOR (i , 0 , ::n/2)

{

if (compId[nodeID(i)] == compId[NOT(nodeID(i))])

return false;

invComp[compId[nodeID(i)]] = compId[NOT(nodeID(i))];

invComp[compId[NOT(nodeID(i))]] = compId[nodeID(i)];

}

adjComp.clear(), adjComp.resize(compnum);

FOR (ii , 0 , ::n)

{

for (int kk = head[ii]; kk != -1; kk = nxt[kk]) {

int jj = to[kk];

int i = compId[ii], j = compId[jj];

if (i == j)

continue;

adjComp[i].pb(j);

}

}

topo();

mem(compres, -1);

FOR (i , 0 , sorSize)

{

int id = sortedOrder[i];

if (compres[id] != -1)

continue;

int invID = invComp[id];

compres[id] = 0, compres[invID] = 1;

}

return true;

}

## Algorithm X

//Code for Sudoku 16\* 16

**#define** fo(i,n) **for**(i=0;i<(n);++i)

**typedef** vector<**int**> vi;

**typedef** vector<string> vs;

**typedef** vector<**double**> vd;

**#define** sz(x) ((**int**)(x).size())

**#define** all(x) x.begin(),x.end()

**#define** pb(x) push\_back(x)

// dancing links with pointers used when only one test case

**#define** MAXROW 16\*16\*16

**#define** MAXCOLS 16\*16\*5

**int** fcol; // fixed constraints

**int** cols; // count of columns (constraints)

vector<vector<**int**> > adj;

**struct** node {

node\* lf, \*rt, \*up, \*dn;

**int** id;

**union** {

node\* hdr;

**int** cnt;

};

**inline** **void** **set**(node\* l, node\* r, node\* u, node\* d, **int** idx, node\* h) {

lf = (l), rt = (r), up = (u), dn = (d), id = (idx), hdr = (h); // el coach 2al mtshlsh el akwas

lf->rt = **this**;

rt->lf = **this**;

up->dn = **this**;

dn->up = **this**;

}

**inline** **void** **coverLR**() {

lf->rt = rt;

rt->lf = lf;

}

**inline** **void** **coverUD**() {

up->dn = dn;

dn->up = up;

}

**inline** **void** **unCoverLR**() {

lf->rt = **this**;

rt->lf = **this**;

}

**inline** **void** **unCoverUD**() {

up->dn = **this**;

dn->up = **this**;

}

**inline** **void** **coverCol**() {

coverLR();

**for** (node\* x = dn; x != **this**; x = x->dn)

**for** (node\* y = x->rt; y != x; y = y->rt) {

y->coverUD();

y->hdr->cnt--;

}

}

**inline** **void** **unCoverCol**() {

**for** (node\* x = up; x != **this**; x = x->up)

**for** (node\* y = x->lf; y != x; y = y->lf) {

y->unCoverUD();

y->hdr->cnt++;

}

unCoverLR();

}

};

node\* root;

**inline** node\* **selectMinC**() {

node\* mn = NULL;

**int** mnCnt = INT\_MAX;

**for** (node\* tmp = root->rt; tmp->id < fcol && tmp != root; tmp = tmp->rt)

**if** (tmp->cnt < mnCnt)

mn = tmp, mnCnt = tmp->cnt;

**return** mn;

}

**int** solCnt;

**int** sol[MAXROW];

**inline** **bool** **algoX**() {

node\* mn = selectMinC();

**if** (!mn)

**return** **true**; // turn into false if all solutions required

mn->coverCol();

**for** (node\* x = mn->dn; x != mn; x = x->dn) {

**for** (node\* y = x->rt; y != x; y = y->rt)

y->hdr->coverCol();

sol[solCnt++] = x->id;

**if** (algoX())

**return** **true**;

solCnt--;

**for** (node\* y = x->lf; y != x; y = y->lf)

y->hdr->unCoverCol();

}

mn->unCoverCol();

**return** **false**;

}

node\* hdrs[MAXCOLS];

**inline** **void** **build**() {

solCnt = 0;

root = **new** node();

root->set(root, root, root, root, 0, 0);

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

hdrs[i] = **new** node();

hdrs[i]->set(root->lf, root, hdrs[i], hdrs[i], i, 0);

}

**for** (**int** i = 0; i < sz(adj); i++) {

node\* fn;

**for** (**int** k = 0; k < sz(adj[i]); k++) {

**int** j = adj[i][k];

**if** (k)

(**new** node())->set(fn->lf, fn, hdrs[j]->up, hdrs[j], i, hdrs[j]);

**else** {

fn = **new** node();

fn->set(fn, fn, hdrs[j]->up, hdrs[j], i, hdrs[j]);

}

hdrs[j]->cnt++;

}

}

}

**inline** **void** **init**(**int** n) {

adj.clear();

adj.resize(n);

}

**char** b[16][17];

**int** **main**() {

**char**\* t = "";

**while** (1) {

**if** (**scanf**(" %c", &b[0][0]) == EOF)

**break**;

**int** cnt = b[0][0] != '-';

**for** (**int** i = 0; i < 16; i++)

**for** (**int** j = i == 0; j < 16; j++)

**scanf**(" %c", b[i] + j), cnt += b[i][j] != '-';

**int** cell = 0;

**int** rws = 16 \* 16;

**int** cls = rws + 16 \* 16;

**int** bxs = cls + 16 \* 16;

**int** fxd = bxs + 16 \* 16;

cols = fcol = fxd + cnt;

init(16 \* 16 \* 16);

**for** (**int** i = 0; i < 16; i++)

**for** (**int** j = 0; j < 16; j++)

**for** (**int** k = 0; k < 16; k++) {

**int** rnk = (i \* 16 + j) \* 16 + k;

adj[rnk].pb(cell+i\*16+j);

adj[rnk].pb(rws+i\*16+k);

adj[rnk].pb(cls+j\*16+k);

**int** bxi = i / 4;

**int** bxj = j / 4;

**int** bi = bxi \* 4 + bxj;

adj[rnk].pb(bxs+bi\*16+k);

**if** (b[i][j] == k + 'A')

adj[rnk].pb(fxd++);

}

build();

algoX();

**for** (**int** l = 0; l < solCnt; l++) {

**int** k = sol[l] % 16;

sol[l] /= 16;

**int** j = sol[l] % 16;

**int** i = sol[l] / 16;

b[i][j] = k + 'A';

}

**printf**(t);

t = "\n";

**for** (**int** i = 0; i < 16; i++)

**printf**("%s\n", b[i]);

}

**return** 0;

}

## Stable Marriage Problem

// wr[w][m] --> the precedence of man number "m" with respect to woman number "w", the less value the more important that man to the woman

vector<vector<**int**> > wr;

// mp[m] --> has the a deque which contains the women in order of importance to this man "m", the first woman is the most important one to this man "m"

vector<deque<**int**> > mp;

// queue of mans, Initially contains all mans indi ces

queue<**int**> unMatchedMen;

// "wm" contains the result, such that wm[w] --> the man index who is married to this woman "w"

// "mw" contains the result, such that mw[m] --> the women index who is married to this man "m"

vector<**int**> wm, mw;

// this algorithm depends on the adaptive greedy approach

**void** **stableMarrageProblem**() {

**while** (unMatchedMen.size()) {

**int** mi = unMatchedMen.front();

unMatchedMen.pop();

**while** (1) {

**int** wi = mp[mi].front();

mp[mi].pop\_front();

**if** (wm[wi] == -1) {

wm[wi] = mi;

mw[mi] = wi;

**break**;

} **else** {

**int** mdi = wm[wi];

**if** (wr[wi][mi] < wr[wi][mdi]) {

wm[wi] = mi;

mw[mi] = wi;

unMatchedMen.push(mdi);

mw[mdi] = -1;

**break**;

}

}

}

}

}

## Bi – Connectivity

#define MAXN 50009

#define MAXE 50009\*2

int head[MAXN], to[MAXE], nxt[MAXE], from[MAXE];

int last;

int n;

void init() {

last = 0;

memset(head, -1, n \* sizeof(head[0]));

}

void addEdge(int f, int t) {

nxt[last] = head[f];

to[last] = t;

from[last] = f;

head[f] = last++;

}

void addBiEdge(int f, int t) {

addEdge(f, t);

addEdge(t, f);

}

// dfsId -> time of visiting a node

// lowId -> low Link of a node

// rootChild -> number of children elli 3amlt mn 3ndohom dfs

// lel root

// dep -> recursion depth

int isArt[MAXN], isBridge[MAXE], dfsId[MAXN], lowId[MAXN];

int dfsIdx, visID, rootId, rootChild;

int vis[MAXN];

int stk[MAXE], stkId, dep, level[MAXN];

vector<vector<int> > Bicomp;

int compID[MAXE], ncomp;

void extractComponentSTL() {

Bicomp.clear();

Bicomp.resize(ncomp);

for (int i = 0; i < last; i += 2)

Bicomp[compID[i]].push\_back(i);

}

void extractComponent(int i) {

do {

compID[stk[--stkId]] = ncomp;

compID[stk[stkId] ^ 1] = ncomp;

} while (stk[stkId] != i);

ncomp++;

}

void bidfs(int u, int rpei) { //Id of reverse of the parent edge

dfsId[u] = lowId[u] = dfsIdx++;

vis[u] = visID;

level[u] = dep++;

for (int i = head[u]; i != -1; i = nxt[i]) {

if (i == rpei)

continue;

int v = to[i];

if (vis[v] != visID) {

stk[stkId++] = i;

bidfs(v, i ^ 1);

if (u == rootId) {

if (++rootChild > 1)

isArt[u] = visID;

extractComponent(i);

}

if (lowId[v] > dfsId[u])

isBridge[i] = isBridge[i ^ 1] = visID;

if (u != rootId && lowId[v] >= dfsId[u])

isArt[u] = visID, extractComponent(i);

lowId[u] = min(lowId[u], lowId[v]);

}

else {

if (level[v] <= level[u] - 1)

stk[stkId++] = i;

lowId[u] = min(lowId[u], dfsId[v]);

}

}

dep--;

}

void Bi() {

visID++;

dfsIdx = 0;

stkId = 0;

ncomp = 0;

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

if (vis[i] != visID) {

rootId = i;

rootChild = 0;

bidfs(i, -1);

}

}

void Print() {

cout << "Articulation points\n";

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

if (isArt[i] == visID)

cout << i << " ";

cout << "\nBridges\n";

for (int i = 0; i < last; i += 2)

if (isBridge[i] == visID)

cout << from[i] << " <-> " << to[i] << "\n";

extractComponentSTL();

int nc = Bicomp.size();

cout << "Number of components is " << nc << endl;

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

cout << "\n com " << i + 1 << endl;

for (int j = 0, k = Bicomp[i][j]; j < int(Bicomp[i].size()) && (k =

Bicomp[i][j]) > -1; j++)

cout << from[k] << " <-> " << to[k] << "\n";

}

}

## Bellman Ford

const int MAXN = 105, MAXE = MAXN \* MAXN;

int head[MAXN], nxt[MAXE], to[MAXE], cost[MAXE], cntEdges;

void init(int n) {

memset(head, -1, sizeof(head[0]) \* n);

cntEdges = 0;

}

void addEdge(int f, int t, int c) {

to[cntEdges] = t;

cost[cntEdges] = c;

nxt[cntEdges] = head[f];

head[f] = cntEdges++;

}

int n, m, dis[MAXN];

int inQ[MAXN], vid = 0;

int Q[MAXN], qf, qe, qs;

bool bellman(int src) {

memset(dis, 0x3f, sizeof(dis[0]) \* n);

dis[src] = 0;

qf = qe = -1;

inQ[src] = ++vid;

qs = 0;

Q[qs++, qe = (++qe % MAXN)] = src;

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

int s = qs;

while (s--) {

int node = Q[qs--, qf = (++qf % MAXN)];

inQ[node] = 0;

for (int k = head[node]; k != -1; k = nxt[k]) {

int j = to[k];

int c = cost[k];

if (dis[j] > dis[node] + c) {

dis[j] = dis[node] + c;

if (inQ[j] != vid)

Q[qs++, qe = (++qe % MAXN)] = j, inQ[j] = vid;

}

}

}

if (!qs)

return true;

}

return false;

}

## Partitioning

//this means that u start value s then s + i\*DIVIDE\_RANGE such that s + i\*DIVIDE\_RANGE < e

//every time the range will be divided by DIVIDE\_RANGE and so on

**#define** DIVIDE\_RANGE 10 //the termination of the delta value once it's almost zero depending of the problem

**#define** TERMINATE 1e-9

**long** **double** **get\_best\_using\_partitioning**(**long** **double** start, **long** **double** end) {

**long** **double** delta = (end - start) / DIVIDE\_RANGE, res = oo;

**long** **double** best;

**while** (delta > TERMINATE) {

**for** (**long** **double** current = start; current <= end; current += delta) {

**long** **double** temp = solve(current);

**if** (temp < res) {

res = temp;

best = current;

}

}

start = best - delta;

end = best + delta;

delta /= DIVIDE\_RANGE;

}

**return** best;

}

## Treap as array (fast insert, delete)

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

typedef struct item \* pitem;

struct item {

int prior, cnt;

bool rev;

pitem l, r;

ll value, sum, lazy, mx, mn;

item(ll v) :

prior(rand()), cnt(1), rev(0), l(0), r(0), value(v), sum(v), lazy(0), mx(

v), mn(v) {

}

};

int cnt(pitem it) {

return it ? it->cnt : 0;

}

ll sum(pitem it) {

return it ? it->sum : 0;

}

ll mx(pitem it) {

return it ? it->mx : LONG\_LONG\_MIN;

}

ll mn(pitem it) {

return it ? it->mn : LONG\_LONG\_MAX;

}

void upd\_cnt(pitem it) {

if (it) {

it->cnt = cnt(it->l) + cnt(it->r) + 1;

it->sum = sum(it->l) + sum(it->r) + it->value;

it->mx = max(it->value, max(mx(it->l), mx(it->r)));

it->mn = min(it->value, min(mn(it->l), mn(it->r)));

}

}

void addLazy(pitem it, ll val) {

it->lazy += val;

it->value += val;

it->sum += val \* it->cnt;

it->mx = max(it->value, max(mx(it->l), mx(it->r)));

it->mn = min(it->value, min(mn(it->l), mn(it->r)));

}

void push(pitem it) {

if (it && it->rev) {

it->rev = false;

swap(it->l, it->r);

if (it->l)

it->l->rev ^= true;

if (it->r)

it->r->rev ^= true;

}

if (it && it->lazy) {

if (it->l)

addLazy(it->l, it->lazy);

if (it->r)

addLazy(it->r, it->lazy);

it->lazy = 0;

}

}

void merge(pitem & t, pitem l, pitem r) {

push(l);

push(r);

if (!l || !r)

t = l ? l : r;

else if (l->prior > r->prior)

merge(l->r, l->r, r), t = l;

else

merge(r->l, l, r->l), t = r;

upd\_cnt(t);

}

void split(pitem t, pitem & l, pitem & r, int key, int add = 0) {

if (!t)

return void(l = r = 0);

push(t);

int cur\_key = add + cnt(t->l);

if (key <= cur\_key)

split(t->l, l, t->l, key, add), r = t;

else

split(t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;

upd\_cnt(t);

}

void reverse(pitem &t, int l, int r) {

pitem t1, t2, t3;

split(t, t1, t2, l);

split(t2, t2, t3, r - l + 1);

t2->rev ^= true;

merge(t, t1, t2);

merge(t, t, t3);

}

void add(pitem &t, int l, int r, ll val) {

pitem t1, t2, t3;

split(t, t1, t2, l);

split(t2, t2, t3, r - l + 1);

addLazy(t2, val);

merge(t, t1, t2);

merge(t, t, t3);

}

ll get(pitem &t, int l, int r) {

pitem t1, t2, t3;

split(t, t1, t2, l);

split(t2, t2, t3, r - l + 1);

ll ret = t2->sum;

merge(t, t1, t2);

merge(t, t, t3);

return ret;

}

pair<ll, ll> rmq(pitem &t, int l, int r) {

pitem t1, t2, t3;

split(t, t1, t2, l);

split(t2, t2, t3, r - l + 1);

pair<ll, ll> ret(t2->mn, t2->mx);

merge(t, t1, t2);

merge(t, t, t3);

return ret;

}

void del(pitem &t, int pos) {

pitem t1, t2, t3;

split(t, t1, t2, pos);

split(t2, t2, t3, 1);

delete t2;

merge(t, t1, t3);

}

void output(pitem t) {

if (!t)

return;

push(t);

output(t->l);

cout << t->value << " ";

output(t->r);

}

void insert(pitem &t, int pos, ll val) {

pitem t1, t2, new\_item = new item(val);

split(t, t1, t2, pos);

merge(t1, t1, new\_item);

merge(t, t1, t2);

}

int main() {

ios::sync\_with\_stdio(0);

cin.tie(NULL);

cout.tie(NULL);

#ifndef ONLINE\_JUDGE

freopen("test.in", "rt", stdin);

// freopen("o.txt", "wt", stdout);

#endif

int n, q;

scanf("%d", &n);

pitem root = 0;

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

int x;

scanf("%d", &x);

insert(root, i, x);

}

scanf("%d", &q);

while (q--) {

int l, r;

scanf("%d%d", &l, &r);

printf("%lld\n", rmq(root, l, r).first);

}

return 0;

}

## Treap

**struct** node {

node \*left, \*right;

**int** value, freq, priority, size;

**static** node\* *sentinel*;

**node**() {

**memset**(**this**, 0, **sizeof** \***this**); //initialize all member variables to zero }

node(**int** v) {

value = v; freq = size = 1;

priority = rand(); left = right = sentinel;}

**void** **update**() {

size = freq + left->size + right->size;

}

}

;

node\* node::sentinel = **new** node();

node\* **rotateRight**(node\* Q) {

node\* P = Q->left;

Q->left = P->right;

P->right = Q;

Q->update();

P->update();

**return** P;

}

node\* **rotateLeft**(node\* P) {

node\* Q = P->right;

P->right = Q->left;

Q->left = P;

P->update();

Q->update();

**return** Q;

}

node\* **balance**(node\* root) {

**if** (root->left->priority > root->priority)

root = rotateRight(root);

**else** **if** (root->right->priority > root->priority)

root = rotateLeft(root);

**return** root;

}

node\* **insert**(node\* root, **int** val) {

**if** (root == node::sentinel)

**return** **new** node(val);

**if** (val == root->value) {

root->freq++;

root->size++;

**return** root;

}

**if** (val < root->value)

root->left = insert(root->left, val);

**else**

root->right = insert(root->right, val);

root->update();

root = balance(root);

**return** root;

}

**int** **lower\_bound**(node\* root, **int** x) { //number of elements less than x in the tree

**if** (root == node::sentinel)

**return** 0;

**if** (x == root->value)

**return** root->left->size;

**return** (x < root->value) ? lower\_bound(root->left, x)

: root->left->size + root->freq + lower\_bound(root->right, x);

}

node\* **remove**(node\* root, **int** v) {

**if** (root == node::sentinel)

**return** root;

**if** (v < root->value)

root->left = remove(root->left, v);

**else** **if** (v > root->value)

root->right = remove(root->right, v);

**else** {

**if** (root->freq > 1) {

root->freq--;

root->size--;

**return** root;

}

**if** (root->left == node::sentinel)

root = root->right;

**else** **if** (root->right == node::sentinel)

root = root->left;

**else** {

**if** (root->left->priority < root->right->priority)

root = rotateRight(root);

**else**

root = rotateLeft(root);

root = remove(root, v);

}

}

root->update();

**return** root;

}

**int** **upper\_bound**(node\* root, **int** x) {

//number of elements less than or equal to x in the tree

**if** (root == node::sentinel)

**return** 0;

**if** (x == root->value)

**return** root->left->size + root->freq;

**return** (x < root->value) ? upper\_bound(root->left, x): root->left->size + root->freq + upper\_bound(root->right, x);

}

**int** **getByIndex**(node\* root, **int** idx) {

**if** (idx < root->left->size)

**return** getByIndex(root->left, idx);

**if** (idx >= root->left->size + root->freq)

**return** getByIndex(root->right,

idx - (root->left->size + root->freq));

**return** root->value;

}

## Expressions and Parsing

//#define \_\_put\_brackets\_in\_tree

**struct** ExpParsing {

**enum** TYPE {

*OP*, *NUM*, *VAR*, *BRAC*, *SEMICOLON*, *LN*, *EOE*

};

**typedef** pair<TYPE, string> TOKEN;

queue<TOKEN> TOKS;

map<string, TYPE> reservedWords; // saving all reserved words

map<string, **int**> vars; // saving all variables with thier values

**void** **reserved**(TOKEN &t) { // take token if its one of reserved words it adapt its type

map<string, TYPE>::iterator it = reservedWords.find(t.second);

**if** (it != reservedWords.end())

t.first = it->second;

}

**void** **Tokinize**(**const** **char**\* exp) { // parsing the statment to tokens

TOKEN t;

**for** (**const** **char**\* c = exp; \*c; c++) {

**if** (**isspace**(\*c))

**continue**;

**switch** (\*c) {

**case** '+':

**case** '-':

**case** '=':

**case** '/':

**case** '\*':

**case** '%':

**case** '^':

t.second = string(1, \*c);

t.first = *OP*;

**break**;

// case ':' : t.second=string(1,\*c); t.first=OP; if(\*(c+1)=='=') { t.second+=\*(++c); }

**case** '(':

**case** ')':

t.second = string(1, \*c);

t.first = *BRAC*;

**break**;

**case** ';':

t.second = string(1, \*c);

t.first = *SEMICOLON*;

**break**;

**default**:

t.second = string(1, \*c);

**if** (**isdigit**(\*c++)) {

t.first = *NUM*;

**while** (**isdigit**(\*c) || \*c == '.')

t.second += \*(c++);

} **else** {

t.first = *VAR*;

**while** (**isalnum**(\*c) || \*c == '\_')

t.second += \*(c++);

}

c--; // on for loop there is c++

reserved(t);

} // if this token is a reserved word it will adapt its type

TOKS.push(t);

}

t.first = *EOE*; // end of expression ( to avoid RTE )

t.second = "";

TOKS.push(t);

}

**struct** NODE { // if (memory limit exceed) Destructor recommended

TOKEN r;

NODE\* lf, \*rt, \*p;// parent

**NODE**() :

lf(0), rt(0), p(0) {

}

**NODE**(TOKEN \_r, NODE\* \_lf, NODE\* \_rt) :

r(\_r), lf(\_lf), rt(\_rt), p(0) {

}

};

// NODE\* expr(); // declration (out of struct)

NODE\* **base**() {

TOKEN t = TOKS.front();

TOKS.pop();

NODE\* n;

**switch** (t.first) {

**case** *NUM*:

**case** *VAR*:

**return** **new** NODE(t, 0, 0);

**case** *BRAC*:

n = expr();

TOKS.pop();

**#ifdef** \_\_put\_brackets\_in\_tree

**return** **new** NODE(make\_pair(BRAC,string("")),n,0);

**#else**

**return** n;

**#endif**

**case** *OP*:

**return** **new** NODE(t, 0, base()); // unary minus

**case** *LN*:

TOKS.pop();

n = expr();

TOKS.pop();

**return** **new** NODE(t, n, 0);

**default**:

;

}

**return** n;

}

NODE\* **factor**() {

NODE\* b = base();

TOKEN t = TOKS.front();

**if** (t.second != "^") {

**return** b;

}

TOKS.pop();

**return** **new** NODE(t, b, factor());

}

NODE\* **term\_**(NODE\* n) {// term'

TOKEN t = TOKS.front();

**if** (t.second == "\*" || t.second == "/" || t.second == "%") {

TOKS.pop();

**return** term\_(**new** NODE(t, n, factor()));

}

**return** n;

}

NODE\* **term**() {

**return** term\_(factor());

}

NODE\* **expr\_**(NODE\* n) { // expr'

TOKEN t = TOKS.front();

**if** (t.second == "+" || t.second == "-") {

TOKS.pop();

**return** expr\_(**new** NODE(t, n, term()));

}

**return** n;

}

NODE\* **expr**() {

**return** expr\_(term());

}

**int** **eval**(NODE\* t) {

**if** (t == 0)

**return** 0;

**int** res;

**switch** (t->r.first) {

**case** *OP*:

**switch** (t->r.second[0]) {

**case** '+':

**return** eval(t->lf) + eval(t->rt);

**case** '-':

**return** eval(t->lf) - eval(t->rt);

**case** '/':

**return** eval(t->lf) / eval(t->rt);

**case** '\*':

**return** eval(t->lf) \* eval(t->rt);

**case** '%':

**return** eval(t->lf) % eval(t->rt);

**case** '^':

**return** (**int**) **pow**((**double**) eval(t->lf), (**double**) eval(t->rt));

}

**case** *NUM*:

**sscanf**(t->r.second.c\_str(), "%d", &res);

**return** res;

**case** *VAR*:

**return** vars[t->r.second];

**case** *BRAC*:

**return** eval(t->lf);

}

**return** 0;

}

**void** **statement**() {

TOKEN var = TOKS.front();

TOKS.pop();

**if** (TOKS.empty())

**return**; // if its empty line

TOKS.pop();

NODE\* tree = expr();

// actual main for vary according to problem statement

//most of expression be on this BNF

//EXP -> TERM E'

//E' -> + TERM E'| - TERM E'

//TERM -> FACTOR T'

//T' -> \* FACTOR T' | / FACTOR T' | e

//FACTOR -> BASE^FACTOR|BASE

//BASE -> VAR| NUM| (EXP)

vars[var.second] = eval(tree);

TOKS.pop(); // for semicolon --Remove it if there is no semicolons--

TOKS.pop();

} // for EOE

NODE\* **deff**(NODE\* t) { // deffrentiation

NODE\* t1, \*t2, \*t3, \*t4, \*t5;

**switch** (t->r.first) {

**case** *OP*:

**switch** (t->r.second[0]) {

**case** '-':

**if** (!t->lf) {

**if** (t->rt->r.first == *NUM*)

**return** **new** NODE(make\_pair(*NUM*, string("0")), 0, 0);

**return** **new** NODE(t->r, 0, deff(t->rt));

}

**case** '+':

**return** **new** NODE(t->r, deff(t->lf), deff(t->rt));

**case** '\*':

t1 = **new** NODE(make\_pair(*OP*, string("\*")), deff(t->lf), t->rt);

t2 = **new** NODE(make\_pair(*OP*, string("\*")), t->lf, deff(t->rt));

t3 = **new** NODE(make\_pair(*OP*, string("+")), t1, t2);

**return** **new** NODE(make\_pair(*BRAC*, string("")), t3, 0);

**case** '/':

t1 = **new** NODE(make\_pair(*OP*, string("\*")), deff(t->lf), t->rt);

t2 = **new** NODE(make\_pair(*OP*, string("\*")), t->lf, deff(t->rt));

t3 = **new** NODE(make\_pair(*OP*, string("-")), t1, t2);

t4 = **new** NODE(make\_pair(*BRAC*, string("")), t3, 0);

t5 = **new** NODE(make\_pair(*OP*, string("^")), t->rt,

**new** NODE(make\_pair(*NUM*, string("2")), 0, 0));

**return** **new** NODE(make\_pair(*OP*, string("/")), t4, t5);

}

**case** *NUM*:

**return** **new** NODE(make\_pair(*NUM*, string("0")), 0, 0);

**case** *VAR*:

**return** **new** NODE(make\_pair(*NUM*, string("1")), 0, 0);

**case** *BRAC*:

**return** **new** NODE(make\_pair(*BRAC*, string("")), deff(t->lf), 0);

**case** *LN*:

t1 = **new** NODE(make\_pair(*BRAC*, string("")), deff(t->lf), 0);

t2 = **new** NODE(make\_pair(*BRAC*, string("")), t->lf, 0);

**return** **new** NODE(make\_pair(*OP*, string("/")), t1, t2);

}

}

string **print**(NODE\* t) {

**if** (!t)

**return** "";

string res;

**switch** (t->r.first) {

**case** *OP*:

**return** print(t->lf) + t->r.second + print(t->rt);

**case** *NUM*:

**case** *VAR*:

**return** t->r.second;

**case** *BRAC*:

**return** "(" + print(t->lf) + ")";

**case** *LN*:

**return** "ln(" + print(t->lf) + ")";

}

}

map<TOKEN, **int**> prec, notass;// for precedence and associativity

**void** **setprec\_Ass**() {

prec[make\_pair(*OP*, string("+"))] = 1;

prec[make\_pair(*OP*, string("-"))] = 1;

prec[make\_pair(*OP*, string("\*"))] = 2;

prec[make\_pair(*OP*, string("/"))] = 2;

notass[make\_pair(*OP*, string("+"))] = 0;

notass[make\_pair(*OP*, string("-"))] = 1;

notass[make\_pair(*OP*, string("\*"))] = 0;

notass[make\_pair(*OP*, string("/"))] = 1;

}

string **printWithOutBraces**(NODE\* t) {

**if** (!t)

**return** "";

**bool** br = 0;

**switch** (t->r.first) {

**case** *OP*:

**if** (t->p && prec[t->p->r] > prec[t->r])

br = 1;

**if** (t->p && prec[t->p->r] == prec[t->r] && t->p->rt == t

&& notass[t->p->r])

br = 1;

**return** (br ? "(" : "") + printWithOutBraces(t->lf) + t->r.second

+ printWithOutBraces(t->rt) + (br ? ")" : "");

**case** *NUM*:

**case** *VAR*:

**return** t->r.second;

}

}

string **printWithOutBracesAfter3aks**(NODE\* t) {

**if** (!t)

**return** "";

**bool** br = 0;

**switch** (t->r.first) {

**case** *OP*:

**if** (t->p && prec[t->p->r] > prec[t->r])

br = 1;

**return** (br ? "(" : "") + printWithOutBracesAfter3aks(t->lf)

+ t->r.second + printWithOutBracesAfter3aks(t->rt)

+ (br ? ")" : "");

**case** *NUM*:

**case** *VAR*:

**return** t->r.second;

}

}

**void** **makeParents**(NODE\* t) {

**if** (t->lf)

makeParents(t->lf), t->lf->p = t;

**if** (t->rt)

makeParents(t->rt), t->rt->p = t;

}

// if you call e3ks, then you must remove the printWithout

**void** **e3ks**(NODE\* n, **int** par\_prec) {

**if** (n->r.first != *OP* || prec[n->r] != par\_prec)

**return**;

**char** \*ops = "+-\*/";

**int** ind = find(ops, ops + 4, n->r.second[0]) - ops;

ind = (ind / 2) \* 2 + !(ind % 2);

n->r.second[0] = ops[ind];

e3ks(n->lf, par\_prec);

} //e3ks(n->rt, par\_prec);

**void** **zabat\_el\_non\_ass**(NODE \*n) { // distribute - and / operators (which are non-associative) on the other operators

**if** (!n)

**return**;

**if** (n->r.second == "-" || n->r.second == "/")

e3ks(n->rt, prec[n->r]);

zabat\_el\_non\_ass(n->lf);

zabat\_el\_non\_ass(n->rt);

}

};

## KD-Tree

**#define** Type **long** **long**

**#define** DIMS 3

**struct** point {

Type a[DIMS];point(Type aa, Type bb, Type cc) {

a[0] = aa;

a[1] = bb;

a[2] = cc;

}

point() {

}

**bool** **operator <**(**const** point& aa) **const** {

**return** a[0] < aa.a[0] || (a[0] == aa.a[0] && a[1] < aa.a[1]) || (a[0]

== aa.a[0] && a[1] == aa.a[1] && a[2] < aa.a[2]);

}

};

set<point> ss;

vector<point> v;

**struct** node;

node\* nil;

**struct** node {

Type di[DIMS];

node\*l, \*r;

**node**() :

l(nil), r(nil) {

}

**node**(Type a, Type b, Type c, node\*left, node\*right) {

di[0] = a;

di[1] = b;

di[2] = c;

l = left;

r = right;

}

};

**int** n;

**struct** cmp {

**static** **int** *d*;

**bool** **operator()**(**const** point&a, **const** point&b) **const** {

**return** a.a[*d*] < b.a[*d*];

}

};

**int** cmp::d = 0;

node\* **build**(**int** st, **int** en, **int** depth) {

**if** (en < st)

**return** nil;

**if** (en == st)

**return** **new** node(v[st].a[0], v[st].a[1], v[st].a[2], nil, nil);

cmp::d = depth % DIMS;

sort(v.begin() + st, v.begin() + en + 1, cmp());

**int** med = (en + st) / 2;

node\*r = **new** node();

r->di[0] = v[med].a[0];

r->di[1] = v[med].a[1];

r->di[2] = v[med].a[2];

r->l = build(st, med - 1, depth + 1);

r->r = build(med + 1, en, depth + 1);

**return** r;

}

point p;

Type **distSq**(node\*cur) {

Type r = 0;

**for** (**int** i = 0; i < DIMS; ++i)

r += (cur->di[i] - p.a[i]) \* (cur->di[i] - p.a[i]);

**return** r;

}

Type mc;

//finds nearest neighbour to point p

**void** **dfs**(node\*cur, Type& mn, **int** depth) {

**if** (cur == nil)

**return**;

Type d = distSq(cur);

**if** (d == mn)

mc++;

**if** (d < mn && !(cur->di[0] == p.a[0] && cur->di[1] == p.a[1] && cur->di[2]

== p.a[2]))

mn = d, mc = 1;

**int** di = depth % DIMS;

**if** (cur->di[di] > p.a[di]) {

dfs(cur->l, mn, depth + 1);

**if** (mn < (cur->di[di] - p.a[di]) \* (cur->di[di] - p.a[di]))

**return**;

dfs(cur->r, mn, depth + 1);

} **else** {

dfs(cur->r, mn, depth + 1);

**if** (mn < (cur->di[di] - p.a[di]) \* (cur->di[di] - p.a[di]))

**return**;

dfs(cur->l, mn, depth + 1);

}

}

## FFT

**typedef** complex<**double**> Complex;

**const** Complex I(0, 1);

**void** **fft**(**double** theta, vector<Complex> &a) {

**int** n = a.size();

**for** (**int** m = n; m >= 2; m >>= 1) {

**int** mh = m >> 1;

**for** (**int** i = 0; i < mh; i++) {

Complex w = exp(i \* theta \* I);

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

**int** k = j + mh;

Complex x = a[j] – a[k];

a[j] += a[k];

a[k] = w \* x;

}

}

theta \*= 2;

}

**int** i = 0;

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

**for** (**int** k = n >> 1; k > (i ^= k); k >>= 1) {

}

**if** (j < i)

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

}

}

**void** **fft**(vector<Complex> &a) {

**int** n = **ceil**(log(a.size()) / log(2));

a.resize(1 << n);

fft(2 \* PI / a.size(), a);

}

**void** **ifft**(vector<Complex> &a) {

**int** n = **ceil**(log(a.size()) / log(2));

a.resize(1 << n);

fft(-2 \* PI / a.size(), a);

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

a[i] /= a.size();

}

**char** a[11001], b[11001], c[22203];

**void** **mul**() {

**int** sa = strlen(a);

**int** sb = **strlen**(b);

**int** sc = sa + sb + 1;

vector<Complex> A(sc), B(sc), C;

**for** (**int** i = sa - 1, j = 0; i >= 0; i--)

A[j++] = a[i] - '0';

**for** (**int** i = sb - 1, j = 0; i >= 0; i--)

B[j++] = b[i] - '0';

fft(A);

fft(B);

C.resize(A.size());

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

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

ifft(C);

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

**int** cr = **round**(C[i].**real**()) / 10;

C[i] = **fmod**(**round**(C[i].**real**()), 10.0);

C[i + 1] += cr;

}

**int** i = C.size() - 1, j;

**while** (i >= 0 && **fabs**(C[i].**real**()) < 1e-9)

i--;

**if** (i < 0) {

c[0] = '0', c[1] = 0;

**return**;

}

**for** (j = 0; i >= 0; j++, i--)

c[j] = **round**(C[i].**real**()) + '0';

c[j] = 0;

}

## Fraction

**#define** ABS(x) ((x)>=0?(x):-(x))

**struct** frac {

**long** **long** n, d;

**frac**(**const** **long** **long**& N, **const** **long** **long** &D = 1) :

n(N), d(D) {

**long** **long** g = gcd(ABS(n), ABS(d));

**if** (!g) {

**this**->n = **this**->d = 0;

**return**;

}

n /= g;

d /= g;

**if** (n == 0)

d = 1;

**if** (d < 0)

n \*= -1, d \*= -1;

**if** (d == 0)

n = 1;

}

**bool** **operator<**(**const** frac &f) **const** {

**return** n \* f.d < d \* f.n;

}

frac **operator\***(**const** frac &f) **const** {

**return** frac(n \* f.n, d \* f.d);

}

frac **operator/**(**const** frac&f) **const** {

**return** frac(n \* f.d, d \* f.n);

}

frac **operator-**(**const** frac &f) **const** {

**return** frac(n \* f.d - d \* f.n, d \* f.d);

}

frac **operator+**(**const** frac &f) **const** {

**return** frac(n \* f.d + d \* f.n, d \* f.d);

}

};

# Other

## Mimimun cycle mean

//Finds the minimimun cycle mean in the graph represented by weight, if no cycle found it returns INF

//Note that the graph represented by weight must be strongly connected (i.e. there is a path

//from each node i to each node j). if it isn't then run the SCC algorithm to find the components

//and then run MMC on each component and take the minimum

//If there is an edge from i to j then weight[i][j] = weight of that edge, else weight[i][j]=INF

//O(n\*m) = O(n^3) where n is the number of nodes and m is the number of edges

**double** **MMC**(vector<vector<**int**> > weight) {

//Initialize

**int** s = 0, k, u, v, n = weight.size(); //n = nodes num

//d[a][b] hwa el distance from 0 to node b using exactly a edges

vector<vector<**int**> > d(n + 1, vector<**int**>(n, INF + 1));

d[0][s] = 0;

//Compute the distances

**for** (k = 1; k <= n; k++)

**for** (v = 0; v < n; v++)

**for** (u = 0; u < n; u++)

**if** (weight[u][v] < INF)

d[k][v] = min(d[k][v], d[k - 1][u] + weight[u][v]);

//Compute lambda using Karp's theorem

**double** lamda = INF;

**for** (u = 0; u < n; u++) {

**double** currentLamda = -1;

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

**if** (d[n][u] < INF && d[k][u] < INF)

currentLamda = max(currentLamda,

1.0 \* (d[n][u] - d[k][u]) / (n - k));

**if** (currentLamda != -1)

lamda = min(lamda, currentLamda);

}

**return** lamda;

}

## Ternary Search

// search within 90 degrees only (Square)

double ternary(double st = 0.0, double end = M\_PI / 2) {

double size = end - st;

for (; size > eps; size = size \* 2 / 3) {

double a = st + size / 3;

double b = st + size \* 2 / 3;

if (f(a) > f(b))

st = a;

}

return st;

}

## Consecutive integers that sum to a given value

**#define** big **long** **long**

vector<pair<big, big> > **whichSums**(big target) {

big n = (-1 + sqrt(1 + 8 \* target)) / 2, i;

vector<pair<big, big> > res;

**for** (i = 1; i <= n; i++) {

**if** (i % 2) {

**if** (target % i == 0)

res.push\_back(

make\_pair(target / i - i / 2, target / i + i / 2));

} **else** **if** ((2 \* target - i) % (2 \* i) == 0)

res.push\_back(

make\_pair((2 \* target - i) / (2 \* i) - i / 2 + 1,

(2 \* target - i) / (2 \* i) + i / 2));

}

**return** res;

}

## Calculating the palindrome substrings

**int** isP[2500][2500]; //2500 is the max string length

string all; //all the text

**int** **isPalin**(**int** start, **int** end) {

**if** (start == end)

**return** isP[start][end] = 1;

**if** (end == start + 1)

**return** isP[start][end] = (all[start] == all[end]) ? 1 : 0;

**if** (isP[start][end] != -1)

**return** isP[start][end];

**if** (all[start] != all[end])

**return** isP[start][end] = 0;

isP[start][end] = isPalin(start + 1, end - 1);

**return** isP[start][end];

}

//MAIN

//memset(isP, -1, sizeof(isP));

//for(int i = 0; i < all.size(); i++)for(int j = i; j < all.size(); j++)isPalin(i,j);

## Permutation Cycles (disjoint cycles)

vector<vector<**int**> > **getCycles**(vector<**int**> vec) {

vector<**bool**> visited(vec.size(), **false**);

vector<vector<**int**> > cycles;

**while** (**true**) {

**int** start = -1, i;

**for** (i = 0; i < vec.size(); i++)

**if** (!visited[i]) {

start = i;

**break**;

}

**if** (start == -1)

**break**;

i = start;

vector<**int**> cycle;

**while** (**true**) {

cycle.push\_back(i);

visited[i] = **true**;

i = vec[i];

**if** (i == start)

**break**;

}

cycles.push\_back(cycle);

}

**return** cycles;

}

## Flatten rectangles

**struct** rect {

**int** lx, ly, ux, uy, color;

**bool** **operator<**(**const** rect& r) **const** {

**return** lx < r.lx || (lx == r.lx && ly < r.ly)

|| (lx == r.lx && ly == r.ly && ux < r.ux)

|| (lx == r.lx && ly == r.ly && ux == r.ux && uy < r.uy);

}

};

**bool** **valid**(rect M) {

**return** (M.ux <= M.lx || M.uy <= M.ly) ? **false** : **true**;

}

vector<rect> **intersect**(vector<rect> vec, rect N) {

set<rect> result;

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

rect M = vec[i];

//N doesn't intersect M

**if** (N.lx >= M.ux || N.ux <= M.lx || N.ly >= M.uy || N.uy <= M.ly) {

result.insert(M);

**continue**;

}

rect r[4] = { { M.lx, M.ly, N.lx, M.uy, M.color }, { N.ux, M.ly, M.ux,

M.uy, M.color }, { max(N.lx, M.lx), N.uy, min(N.ux, M.ux), M.uy,

M.color }, { max(N.lx, M.lx), M.ly, min(N.ux, M.ux), N.ly,

M.color } };

**for** (**int** j = 0; j < 4; j++)

**if** (valid(r[j]))

result.insert(r[j]);

}

result.insert(N);

vector<rect> v;

**for** (set<rect>::iterator itr = result.begin(); itr != result.end(); itr++)

v.push\_back(\*itr);

**return** v;

}

vector<rect> **flatten**(vector<rect> vec) {

vector<rect> result;

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

result = intersect(result, vec[i]);

**return** result;

}

## Letter tree

**const** **int** MAX = 128; //if MAX is big range but not all values are used, use a map instead of static array

**struct** tree {

tree\* child[MAX];

**tree**() {

**memset**(child, 0, **sizeof**(child));

}

**void** **insert**(vector<**int**>& vec, **int** index) {

**if** (index == vec.size())

**return**;

**if** (child[vec[index]] == 0)

child[vec[index]] = **new** tree();

child[vec[index]]->insert(vec, index + 1);

}

**int** **count**() {

**int** c = 0;

**for** (**int** i = 0; i < MAX; i++)

**if** (child[i] != 0)

c += child[i]->count();

**return** c + 1;

}

};

//MAIN

//tree t;

//t.insert(vec, 0)

//int c = t.count

## Letter Tree(Hashing)

edge edges[maxE]; // memseted with -1

**bool** isLeaf [maxN];//memseted with 0

edge& **getEdge**(**int** ind,**unsigned** **char** c)

{

**int** i = ((ind<<8)+c)%maxE; // da el hashing

**while**(edges[i].from!=-1)

{

**if**(edges[i].from == ind && edges[i].c == c) **break**;

i = ++i%maxE;

}

**return** edges[i];

}

**void** **insert**(**const** **char**\* str,**int** ind = 0)

{

**if**(!\*str)

{

isLeaf[ind] = **true**;

**return** ;

}

edge& e = getEdge(ind,\*str);

**if**(e.from == -1)

{

e.from = ind;

e.to = cN ++;

e.c = \*str;

}

insert(str+1,e.to);

}

**bool** **traverse**(**const** **char**\* str,**int** ind = 0)

{

**if**(!\*str)

{

**return** isLeaf[ind] ;

}

edge& e = getEdge(ind,\*str);

**if**(e.from == -1)

{

**return** **false**;

}

**return** traverse(str+1,e.to);

}

## Letter Tree(Hashing-using hashmap)

**#include**<ext/hash\_map>

**using** **namespace** \_\_gnu\_cxx;

//#define MAXNODES 1000000

**int** nNodes = 1; // root

**struct** hashh

{

**int** **operator()**(**const** pair<**int**,**char**> &p)**const**

{

**return** p.first\*31+p.second;

}

};

hash\_map<pair<**int**,**char**>,**int** ,hashh> edges; //from , char, to

//bool isLeaf[MAXNODES];

vector<**bool**> isLeaf(1);

vector<multiset<**int**> > crab; // fot dfs implementations

**void** **insert**(**const** **char**\* str)

{

**int** cur = 0;

**for**(**const** **char**\* s = str ; \*s ; s++)

{

hash\_map<pair<**int**,**char**>,**int** ,hashh>::iterator it;

it = edges.find(make\_pair(cur,\*s-'a'));

**if**(it==edges.end())

isLeaf.push\_back(0),cur = edges[make\_pair(cur,\*s-'a')]= nNodes++;

**else**

cur = it->second;

}

isLeaf[cur]=1;

}

**bool** **find**(**const** **char**\* str)

{

**int** cur = 0;

**for**(**const** **char**\* s = str ; \*s ; s++)

{

hash\_map<pair<**int**,**char**>,**int** ,hashh>::iterator it;

it = edges.find(make\_pair(cur,\*s-'a'));

**if**(it==edges.end())

**return** **false**;

**else**

cur = it->second;

}

**return** isLeaf[cur];

}

// dfs on the tree

**int** **dfs**(**int** cur)

{

hash\_map<pair<**int**,**char**>,**int** ,hashh>::iterator it;

**int** ret = -(1<<28);

**if**(isLeaf[cur])ret = 0;

**for**(**int** c=0;c<26;c++)

{

it=edges.find(make\_pair(cur,c));

**if**(it==edges.end())

**continue**;

**if**(crab[c].size())

{

**int** x = \*crab[c].rbegin();

crab[c].erase(--crab[c].end());

ret = max(ret,x+dfs(it->second));

crab[c].insert(x);

}

}

**return** ret;

}

## next\_permuatation in java

void next\_permutation(int[] arr) {

int N = arr.length;

int i = N - 1;

while(arr[i-1] >= arr[i]) i = i-1;int j = N;while(arr[j-1] <= arr[i-1]) j = j-1;int temp = arr[i-1];arr[i-1] = arr[j-1];arr[j-1] = temp;i++; j = N;

while (i < j) {temp = arr[i-1];arr[i-1] = arr[j-1];arr[j-1] = temp;i++;j--;}}

}

## Permutations

**int** **getIndex**(**char** \* str) {

**int** res = 0;

**if** (!\*str)

**return** 1;

**bool** vis[26] = { 0 };

**for** (**char** \* s = str + 1; \*s; s++)

**if** (!vis[\*s - 'a'] && \*s < \*str) {

vis[\*s - 'a'] = 1;

**int** count[26] = { 0 };

**int** chars[26];

**int** size = 0, len = 0;

**for** (**char** \* ss = str; \*ss; ss++) {

**if** (ss == s)

**continue**;

**if** (!(count[\*ss - 'a']++))

chars[size++] = \*ss - 'a';

len++;

}

**long** **long** f = 1;

**for** (**int** i = len; i > 1; i--) {

f \*= i;

**for** (**int** j = 0; j < size; j++) {

**int** & r = count[chars[j]];

**while** (r > 1 && f % r == 0) {

f /= r;

r--;

}

}

}

res += f;

}

**return** res + getIndex(str + 1);

}

**typedef** vector<**int**> vi;

// p should contain numbers (0)->(n-1)

// returns the permutation number of p (0 indexed)

**int** **permToIndex**(vi p) {

**if** (sz(p) <= 1)

**return** 0;

**if** (sz(p) == 2)

**return** p[0];

**int** f = 1;

**for** (**int** i = 1; i < sz(p); i++)

f \*= i;

vi r = p;

r.erase(r.begin());

**for** (**int** i = 0; i < sz(r); i++)

**if** (r[i] > p[0])

r[i]--;

**return** f \* p[0] + permToIndex(r);

}

**#define** pb push\_back

// j is the permutaion number

// d is the number of elements in the permutaion

// returns the jth permutaion

vi **indexToPerm**(**int** j, **int** d) {

**if** (d == 1) {

vi ret;

ret.pb(0);

**return** ret;

}

**int** f = 1;

**for** (**int** i = 2; i < d; i++)

f \*= i;

vi r(d);

r[0] = j / f;

vi t = indexToPerm(j % f, d - 1);

**for** (**int** i = 0; i < sz(t); i++)

**if** (t[i] >= r[0])

t[i]++;

**int** ff = 0, tt = 1;

rep(i,sz(t))

r[tt++] = t[ff++];

**return** r;

}

## Date

**bool** **isLeap**(**int** year) {

**return** (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;

}

**int** days[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

**struct** date {

**int** year, month, day;

**date**() {

}

**date**(**int** dd, **int** mm, **int** yy) {

year = yy;

month = mm;

day = dd;

}

**bool** **operator <**(**const** date &d) **const** {

**if** (year != d.year)

**return** year < d.year;

**if** (month != d.month)

**return** month < d.month;

**return** day < d.day;

}

**bool** **operator>**(**const** date &d) **const** {

**if** (year != d.year)

**return** year > d.year;

**if** (month != d.month)

**return** month > d.month;

**return** day > d.day;

}

**bool** **operator <=**(**const** date &d) **const** {

**if** (year != d.year)

**return** year < d.year;

**if** (month != d.month)

**return** month < d.month;

**return** day <= d.day;

}

**bool** **operator>=**(**const** date &d) **const** {

**if** (year != d.year)

**return** year > d.year;

**if** (month != d.month)

**return** month > d.month;

**return** day >= d.day;

}

**bool** **operator ==**(**const** date &d) **const** {

**return** year == d.year && month == d.month && day == d.day;

}

**void** **next**() {

**int** dd = days[month];

**if** (month == 2 && isLeap(year))

dd++;

day++;

**if** (day > dd) {

month++;

day = 1;

**if** (month > 12) {

year++;

month = 1;

}

}

}

**void** **prev**() {

day--;

**if** (day < 1) {

month--;

**if** (month < 1) {

year--;

month = 12;

}

day = days[month];

**if** (month == 2 && isLeap(year))

day++;

}

}

string **toString**() {

stringstream S;

S << day << "/" << month << "/" << year;

**return** S.str();

}

};

## Solving defragmentation problem using segment trees

**const** **int** SIZE = 200000; // 2\*( 1<< ((int)(log2(50000))+1) );

**struct** node {

**int** from, to; //segment this node is responsible for

**int** left, right, big; //size of left, right, biggest spaces with segment

**int** state; //1 for empty, 0 for full, 2 for mixed

} nodes[SIZE];

**int** N, M, MAX\_NODE = 0;

**void** **createTree**(**int** node, **int** from, **int** to) {

nodes[node].from = from;

nodes[node].to = to;

nodes[node].state = 1;

nodes[node].big = nodes[node].right = nodes[node].left = to - from + 1;

MAX\_NODE = max(MAX\_NODE, node);

**if** (from == to)

**return**; //leaf

createTree(2 \* node, from, (from + to) / 2);

createTree(2 \* node + 1, (from + to) / 2 + 1, to);

}

**int** **query**(**int** node, **int** size) {

**if** (nodes[node].big < size)

**return** 0;

**if** (nodes[node].left >= size)

**return** nodes[node].from;

**if** (nodes[2 \* node].big >= size)

**return** query(2 \* node, size);

**if** (nodes[2 \* node].right + nodes[2 \* node + 1].left >= size)

**return** nodes[2 \* node].to - nodes[2 \* node].right + 1;

**return** query(2 \* node + 1, size);

}

**void** **propagateState**(**int** node) {

nodes[2 \* node].state = nodes[2 \* node + 1].state = nodes[node].state;

nodes[2 \* node].left = nodes[2 \* node].right = nodes[2 \* node].big =

nodes[node].state \* (nodes[2 \* node].to - nodes[2 \* node].from + 1);

nodes[2 \* node + 1].left = nodes[2 \* node + 1].right =

nodes[2 \* node + 1].big = nodes[node].state

\* (nodes[2 \* node + 1].to - nodes[2 \* node + 1].from + 1);

}

**void** **modify**(**int** node, **int** from, **int** to, **int** val) {

**if** (nodes[node].from > to || nodes[node].to < from)

**return**;

**if** (nodes[node].from >= from && nodes[node].to <= to) {

nodes[node].state = val;

nodes[node].big = nodes[node].left = nodes[node].right = val

\* (nodes[node].to - nodes[node].from + 1);

**return**;

}

**if** (nodes[node].state != 2) //Make sure children are consistent with me if i'm not mixed

propagateState(node);

modify(2 \* node, from, to, val);

modify(2 \* node + 1, from, to, val);

nodes[node].state =

nodes[2 \* node].state != nodes[2 \* node + 1].state ?

2 : nodes[2 \* node].state;

nodes[node].left =

nodes[2 \* node].state != 1 ?

nodes[2 \* node].left :

nodes[2 \* node].left + nodes[2 \* node + 1].left;

nodes[node].right =

nodes[2 \* node + 1].state != 1 ?

nodes[2 \* node + 1].right :

nodes[2 \* node + 1].right + nodes[2 \* node].right;

nodes[node].big = max(nodes[2 \* node].big, nodes[2 \* node + 1].big);

nodes[node].big = max(nodes[node].big,

nodes[2 \* node].right + nodes[2 \* node + 1].left);

}

## Quad Tree

/\*

Very useful in many cases. One case is compressing of binary image.

Imagine we partition a grid into 4 sections, if there is a region full of same color [0, 1]

we do not need to process that region again.

\*/

**struct** QuadTree {

**bool** isMixed;

**int** val;

QuadTree\* childs[4];

QuadTree() : isMixed(1) {}

QuadTree(**int** v) : isMixed(0), val(v) {}

QuadTree\* getChild(**int** i) {

**if**(isMixed) **return** child[i];

**return** **this**;

}

};

/\*

In comparing 2 trees, one tree may not have same structure as second one.

One nice trick to make them seems similar, is define get function.

\*/

## String utilities

**bool** **isVowel**(**char** t)

{

t = **tolower**(t);

**if**(t == 'a' || t == 'i' || t == 'u' || t == 'o' || t == 'e') **return** **true**;

**return** **false**;

}

string **toLower**(string t)

{

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

{

t[i] = **tolower**(t[i]);

}

**return** t;

}

**bool** **replace**(string& str, string fr, string to)

{

**int** pos;

**if** ((pos = str.find(fr)) != -1) {

str = str.substr(0, pos) + to + str.substr(pos + fr.length());

**return** **true**;

}

**return** **false**;

}

vector<string> **split**(string t, **char** c)

{

string m = "";

vector<string> res;

**for**(**int** i = 0 ; i < sz(t) ; i ++)

{

**if**(t[i] == c&&m!="")

{

res.push\_back(m);

m = "";

}

**else**

m+=t[i];

}

**if**(m!="")

res.push\_back(m);

**return** res;

}

string **toUpper**(string t)

{

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

{

t[i] = **toupper**(t[i]);

}

**return** t;

}

**int** **toDecimal**(string s, **int** base)

{

**int** v, i, result = 0;

**for**(i = 0 ; i < s.size() ; i++)

{

**if**(s[i]>='0' && s[i] <= '9') v = s[i] - '0';

**else** v = s[i]-'A'+10;

result = result\*base+v;

}

**return** result;

}

**int** **StoI**(string s)

{

**int** v, i, result = 0;

**for**(i = 0 ; i < s.size() ; i++)

{

v = s[i] - '0';

result = result\*10+v;

}

**return** result;

}

string toBase(**int** num, **int** base)

{

**if**(num ==0) **return** "0";

string str;

**while**(num!=0)

{

**int** nlet = num%base;

num/= base;

**if**(nlet<0)//for negative base

num++,nlet+=(-1\*base);

**if**(nlet<10) str += (nlet+'0');

**else** str += (nlet-10+'A');

}

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

**return** str;

}

string **ItoS**(**int** num )

{

**if**(num == 0) **return** "0";

string str;

**while**(num!=0)

{

**int** nlet = num%10;

str += (nlet+'0');

num/= 10;

}

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

**return** str;

}

## Int utilities

**double** **dis**(**int** x1,**int** y1,**int** x2,**int** y2)

{

**return** **sqrt**(pow((**double**)**abs**(x1-x2),2)+pow((**double**)**abs**(y1-y2),2));

}

**int** **gcd** (**int** x,**int** y)

{

**if**(y==0)**return** x; **return** gcd(y,x%y);

}

**int** **lcm** (**int** x,**int** y)

{

**return** x/gcd(x,y)\*y;

}

**int** oo = 1000000001;

**int** C[203][203];

**void** **buildnCr**(**int** n) {

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

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

C[i][j] = (j == 0) ? 1 : ( (i == 0) ? 0 : C[i-1][j-1]+C[i-1][j]);

}

## Binary Search using for loop

**bool** **f**(**double**);

**#define** EPS 1e-9

**double** **binarySearch**(**double** s,**double** m)

{

**for**(;m>EPS;m\*=.5)

**if**(f(s+m))// if true take the right part

s+=m;

**return** s;

}

## merge vector of pairs

( remove intersection ) make larger pairs .. v must contain 1 element at least

**void** **merge**(vector<pair<**double**, **double**> >& v, vector<pair<**double**, **double**> >& ans) {

ans.clear();

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

pair<**double**, **double**> cur = v[0];

**for**(**int** i=1;i<v.size();i++)

**if**(v[i].first >= cur.first && v[i].first <= cur.second)

cur.second = max(cur.second, v[i].second);

**else** ans.push\_back(cur), cur = v[i];

ans.push\_back(cur);

}

## Loop on all subsets of 1s for a certain number s

**for**(**int** i=s;i;i=(i-1)&s);

## kthRoot

ll **kthRoot**(ll n, ll k) // return integer kth root for n

{ // Also can be done by binary search for accurate results

**double** root = **pow**((**double**)n, 1.0 / (**double**) k); // will have percision errors

ll realRoot = (ll)(root-1);

**while**(1) {

ll a = realRoot + 1, p = 1;

**for**(**int** j = 0; j < k; j++) // compute a^k

{

**if**(p > n / a) // we exceed n, this also detect overflow

**return** realRoot;

p \*= a;

}

++realRoot;

}

}

## numDigits 1000 has four digits

**int** **numDigits**(**int** n) {

**return** (**int**)**log10**(n)+1;

}

## Roll die

////////////////////

string dir = "NSEW"; // you can rotate a die in 4 directions

//0=top 1=bottom 2=left 3=right 4=front 5=back

**int** rot[][6] = {

// roll ON y-axix

{4, 5, 2, 3, 1, 0}, // N

{5, 4, 2, 3, 0, 1}, // S

// roll ON x-axix

{2, 3, 1, 0, 4, 5}, // E

{3, 2, 0, 1, 4, 5}, // W

// move AROUND z-axix

{0, 1, 5, 4, 2, 3},

{0, 1, 4, 5, 3, 2}

};

string **roll**(string die, **char** d) { // assume d in dir

string ndie = "";

**int** idx = (**int**)dir.find( **toupper**(d) );

lp(i, 6) ndie += die[ rot[idx][i] ];

**return** ndie;

}

// u should in paper, determine how is initial die, E.g. 163452

// not each two faces sum = 7

//////////////////////////////////

// Generate all rotation of a Die

**int** rotLEFT[]={0,1,4,5,3,2};

**int** rotDOWN[]={4,5,2,3,1,0};

**void** **rotate**(string s, set<string> & rots) {

**if** (rots.find(s) != rots.end()) **return**;

rots.insert(s);

string rot1 = "", rot2 = "";

rep(i, 6) rot1 += s[rotLEFT[i]];

rep(i, 6) rot2 += s[rotDOWN[i]];

rotate(rot1, rots);

rotate(rot2, rots);

}

// dice is 6 faces E.g. 012345 [top, bottom, left, right, front back]

string **getNormalDiceForm**(string die) {

set<string> allRotations;

rotate(die, allRotations); // generate all, and take first

**return** \*(allRotations.begin());

}

## Time to string

string **toTime**(**int** total\_sec) //120 sec is 2 minutes

{

**int** days = total\_sec / (60\*60\*24);

**int** hours = total\_sec / (60\*60)-days\*24;

**int** minutes = (total\_sec / 60) % 60;

**int** sec = total\_sec % 60;

string period = " AM";

**if**(hours > 12) hours -= 12, period = " PM";

**return** toStr(days, 2) +':'+ toStr(hours, 2)+':' +

toStr(minutes, 2) +':'+ toStr(sec, 2) + period;

}

## Month names

string months[12] = {"JANUARY", "FEBRUARY", "MARCH", "APRIL", "MAY", "JUNE", "JULY", "AUGUST",

"SEPTEMBER", "OCTOBER", "NOVEMBER", "DECEMBER"};

## Number names and fromNumTOWords and fromWordsTONum

string nums[20] = {

"", "one", "two", "three", "four", "five", "six", "seven",

"eight", "nine", "ten", "eleven", "twelve","thirteen",

"fourteen","fifteen","sixteen","seventeen","eighteen","nineteen"

};

string tenths[10] = {

"", "", "twenty", "thirty", "forty", "fifty", "sixty",

"seventy", "eighty", "ninety"

};

string **fromNumTOWords**(**int** num) //10 is ten

{

**if**(num == 0) **return** "zero";

string res = "", thos = "", hund = "", tens = "";

**if**(num < 0) num \*= -1, res += "negative";

**int** nThousands = num/1000;

**int** nHundreds = (num%1000)/100;

**int** nTenths = num - 1000\*nThousands-100\*nHundreds;

**if**(nThousands) thos += fromNumTOWords(nThousands)+ " thousand";

**if**(nHundreds) hund += nums[nHundreds] + " hundred";

**if**(nTenths) tens = (nTenths < 20 ) ? nums[nTenths] :

tenths[nTenths/10]+' '+nums[nTenths%10];

res += thos + (( (nThousands) ? " " : "")) + hund;

res += ( (nThousands||nHundreds) && nTenths) ? " and " : "";

**return** res +tens;

}

**int** **fromWordsTONum**(string line)//ten is 10

{

map<string, **int**> value;

**for**(**int** i=1; i<20;i++) value[ nums[i] ] = i;

**for**(**int** j=2; j<10;j++) value[ tenths[j] ] = 10\*j;

value["zero"] = 0, value["hundred"] = value["thousand"] = -1;

string word;

**int** answer = 0, tens = 0, negative = 0;

istringstream iss(line);

**while** (iss >> word)

{

**if** (word == "and") **continue**;

**else** **if** (word == "negative") negative = 1;

**else** **if** (value[word] == -1)

{

**if**(word == "thousand")

answer = 1000\*(answer+tens), tens = 0;

**else**

answer += 100\*tens, tens = 0;

}

**else**

tens += value[word];

}

**return** (negative) ? (answer+tens)\*-1 : (answer+tens);

}

## st for 1 21 31, nd for 2 22, rd for 3 23

string **formatPostfix**(**int** n) {

**int** temp, mod1, mod2;

temp = n, mod1 = temp%10, temp/=10, mod2 = temp%10;

**if**(mod2 == 1) **return** "th";

**if**(mod1 == 1) **return** "st";

**if**(mod1 == 2) **return** "nd";

**if**(mod1 == 3) **return** "rd";

**return** "th";

}

## Return angle from hour hand to minute hand.

**double** **clockAngle**(**int** h, **int** m, **int** s = 0) {

**double** exactM = m+s/60.0, exactH = h%12+exactM/60.0; // 60 sec is 1 min, 30 sec is 0.5 min

**double** mDeg = exactM\*6.0; // calc angle clockwise. Each minute is 360/60=6 degree

**double** hDeg = exactH\*30.0; // calc angle clockwise. Each hour is 360/12=30 degree

**if**(hDeg <= mDeg) **return** mDeg-hDeg; // Draw. Simply it is difference

**return** 360 - (hDeg-mDeg); // Draw. Simply it is complement

}

## add "1234" + "56546" = "57780" given base

string B = "0123456789ABCDEF";

**int** **I**(**char** c) { **return** B.find(c); }

string **add**(string a, string b, **int** base) {

**int** mx = max(sz(a), sz(b));

**int** C[200] = {0};

**while**( sz(a) != mx) a = "0" + a;

**while**( sz(b) != mx) b = "0" + b;

reverse( all(a) );

reverse( all(b) );

**for** (**int** i = 0; i < mx; ++i) {

**int** t = C[i] + I(a[i]) + I(b[i]);

C[i] = t % base, C[i+1] += t / base;

}

**int** i = mx;

**while**(i > 0 && C[i] == 0) i --;

string ret = "";

**for** (**int** j = i; j >= 0; --j) ret += B[ C[j] ];

**return** ret;

}

## decToBase

string **decToBase**(ll number, **int** base)

{

**if**(number == 0) **return** "0";

string res = "", encode = "0123456789ABCDEF";

**while**(number)

res = encode[number % base] + res, number /= base;

**return** res;

}

## toDecimal

ll **toDecimal**(string number, **int** base) { // Watchout OVERFLOW inputs

string decode = "0123456789ABCDEF";

ll res = 0;

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

res \*= base, res += decode.find(number[i]);

**return** res;

}

## roman\_to\_int

**int** **value**(**char** ch) {

**if**(ch =='I') **return** 1; **if**(ch =='V') **return** 5;

**if**(ch =='X') **return** 10; **if**(ch =='L') **return** 50;

**if**(ch =='C') **return** 100; **if**(ch =='D') **return** 500;

**return** 1000;

}

**int** **roman\_to\_int**(string roman) {

**int** i, num = 0, len = roman.size()-1;

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

{

**if**(value(roman[i]) >= value(roman[i+1]))

num += value(roman[i]);

**else**

num -= value(roman[i]);

}

num += value(roman[i]);

**return** num;

}

## int\_to\_roman

string **int\_to\_roman**(**int** num) {

// bool valid = (num <= (4999||3999) && int\_to\_roman() == roman\_to\_int()

string roman[] = //Largest integer possible 4999, some people 3999

{ "","I","II","III","IV","V","VI","VII","VIII","IX" //1,2,3,4,..

,"X","XX","XXX","XL","L","LX","LXX","LXXX","XC" //10,20,30,..

,"C","CC","CCC","CD","D","DC","DCC","DCCC","CM" //100,200,300

,"M","MM","MMM","MMMM" //1000,2000...

}; //2222=2000+200+20+2 = MMCCXXII // 4444=MMMMCDXLIV

string roman\_number = "";

**int** i, j, arr[50] = {0|;

**for**(i=0; num; i++) //cut it to thousands, hundreds, tens..

{

**if**(num%10 != 0) arr[i] = i\*9 + (num%10);

num /= 10;

}

**for**(j=0;j<i;j++)

roman\_number = roman[arr[j]] + roman\_number;

**return** roman\_number;

}

## Josephus

// Assume cycle [1 - n], and we kill mth, then 2mth..

// all sent arguments are 1-based

**int** **joseph\_lastKilled**(**int** n, **int** m, **int** firstKilled = 1) {

**int** k = 0;

**for**(**int** i = 2; i<=n; k=(k+m)%i, i++); // k represent last killed person when cycle length=i

k = (k-(m-firstKilled)+10000\*n)%n; // shift the k, note: M may be > n

**while**(k < 0) k += n;

**return** k;

}

**int** **JosephCycle**(**int** n, **int** m, **int** k) // using segment tree

{ // JosephCycle(5, 2, 3) = 5, after how many iter, k will die

**int** cur = 1;

build(1, n,1 ); // build tree from 1-n

**for**(**int** i = n;i > 0; i--) // UNTILL i > 0

{

cur=(cur+m-1)%i;

**if**(cur == 0) cur = i; // I think this is done because it is 1-based

// cur the index to be killed starting from START.

**if**( del(1, n, cur, 1) == k ) **return** n-i+1;

}

**return** -1; // must not happen

}

// test if any element in range n/2 is killed in n/2 iteration

**bool** **JosephCycleTest**(**int** n, **int** m) { // test first n/2 kill operation

**for**(**int** cur = 0, i = n; i >n/2 ; i--) {

cur = (cur+m-1)%i;

**if**(cur < n/2) **return** **false**; // 0-based compare parameters

}

**return** **true**;

}

## grayCode

**int** **inverseGray**(**int** n) {

**int** ish = 1, ans = n;

**while**(**true**) {

**int** idiv = ans >> ish;

ans ^= idiv;

**if** (idiv <= 1 || ish == 32) **return** ans;

ish <<= 1;

}

}

**void** **grayCode**(**int** n) {

lp(i, 1<<n)

cout<<(i^(i>>1))<<"\n";

}

## stirling1

// number of permutations of n elements with k permutation cycles.

// E.g perm(1, 2, 3, 4) = 2, 1, 4, 3 has 2 cycles. {1, 2} , {3, 4}

ll **stirling1**(ll n, ll k) {

**if**(k == 0) **return** n == k;

**if**(n == 0) **return** 0;

**return** (n-1) \* stirling1(n-1, k) + stirling1(n-1, k-1);

}

## stirling2

// number of ways to partition a set of n elements into k groups.

// E.g. set{1, 2, 3, 4, 5} can be partioned to {1, 3, 5} {2, 4}

ll **stirling2**(ll n, ll k) {

**if**(n == k || k == 1) **return** 1;

**return** k \* stirling2(n-1, k) + stirling2(n-1, k-1);

}

## build\_bellNumbers

const ll MAX\_BELL = 1000;

ll bell[MAX\_BELL] = {1};

ll rows[2][MAX\_BELL] = {1}, p = 0;

// number of partitions of a set of size n

// E.g. set{1, 2, 3, 4} can be divided {{1}, {3,2, 4}} or {{2, 4} {1, 3}}

// NOTE: partitions {{1, 2},{3, 4}} and {{3, 4}, {2, 1}} are counted once. NO ORDER ISSUES

**void** **build\_bellNumbers**() { // O(n^2)

build\_nCk();

for(i, 1, MAX\_BELL) repi(k, 0, i) bell[i] += C[i-1][k] \* bell[k];

}

**void** **build\_bellNumbers2**() { // O(n\*(n+1)/2) // bell triangle

repi(i, 1, MAX\_BELL) {

p = !p, bell[i-1] = rows[p][0] = rows[!p][i-1];

repi(j, 1, i+1) rows[p][j] = rows[p][j-1] + rows[!p][j-1];

}

}

## num\_digits\_of\_n\_combination\_k

**int** **num\_digits\_of\_n\_combination\_k**(**int** n, **int** k) {

**double** comb = 0;

**if**(k > n/2) k = n-k;

**int** i, j = k;

**for**(i=n;i>n-k;i--) {

comb += **log10**(i);

**for**(;j>0;j--) {

**if**(comb < 0) **break**;

comb -= **log10**(j);

}

}

**return** (**int**)(**floor**(comb)+1);

}

## fast\_Fibonacci O(log(n))

**int** **fast\_Fibonacci**(**int** n){

**int** i=1, h=1, j=0, k=0, t;

**while** (n > 0) {

**if** (n%2 == 1)

t = j\*h, j = i\*h + j\*k + t, i = i\*k + t;

t = h\*h, h = 2\*k\*h + t, k = k\*k + t, n = n/2;

}

**return** j;

/\* Golden Mean

double d = sqrt(5);

double b=pow( (1+d)/2, n);

double c=pow( (1-d)/2, n);

cout<<(b-c)/d;

\*/

}

## repeating\_digits\_after\_decimal\_point\_from\_rational\_number

**int** **numBeforeRepeat**(**int** n, **int** d) {

**int** c2=0, c5=0;

**if** (n == 0) **return** 1;

**while** (d%2==0) d/=2, c2++;

**while** (d%5==0) d/=5, c5++;

**while** (n%2==0) n/=2, c2--;

**while** (n%5==0) n/=5, c5--;

**if** (c2 > c5)

**return** c2 > 0 ? c2 : 0;

**return** c5 > 0 ? c5 : 0;

}

**void** **repeating\_fractions\_from\_rational\_mumber**(**int** n, **int** d)

{ // you can apply it, to any base, but keep n, d in decimal base

cout<<n/d<<'.', n%=d;

**int** m=numBeforeRepeat(n,d);

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

n\*=10, cout<<n/d, n%=d;

**int** count = 0, r = n;

**if**(r!=0)

{

**do**

{

n\*=10, cout<<n/d, n%=d, count++;

} **while** (n!=r);

cout<<"\nThe last "<<count<<" digits repeat forever";

}

}

## CONTEST STRATEGY

REGIONALS:

1- Sort problem set by length and assign to members starting with the fastest

2- Read the problem CAREFULLY, if it is an ACE code it directly on PC and go to step 8

3- Give yourself 5 minutes of thinking even if the problem is hard, you only need to understand the problem statement very well and think in a solution if possible

4- Describe the problem to the person who is better at the problem area, whom should listen very carefully and make sure he understands the problem very well,

5- This small meeting should decide one of the following: 1-the problem should be delayed 2-you should write the solution you came up with 3-you both stay for sometime thinking in a solution 4-you only should stay for sometime thinking in a solution

6- If you both decided that this problem is to be solved, the better of the two at the problem area will read the problem carefully (if not yet) write code on paper and get approval from the backup that the code is COMPLETE

7- Once the PC is free, copy ur code there, make sure you copy the input correctly from the problem statement and debug for the first 10 minutes to match the sample output, if more debugging is needed the backup should join for another 10 minutes, if still print and debug on paper

8- if you submit and got WA or TLE or RTA review the checklist, read the problem again, debug on paper or whatever for another 20 minutes, if u found bug(s)interrupt the man on the PC, write the testcase u suspect, run and make sure u get WA, then take backup of the code apply ur fix, run and make sure the output is correct and submit

9- if the offline debug took 20 minutes the backup should read the problem and the code and spend 10 minutes with you, if u couldn't get it leave the problem immediately and get back to it later

10- In the last hour don't start a new problem (unless you've no wrong submissions), sort problems by most solved and for each wrong submission the author and the backup (and the third if he isn't doing anything) should debug it.

## HINTS FOR THE CONTEST

Hints: -Compete with problemset instead of team, use score board only to know which problems are solved

WA bugs:

-CHECK THE SPELLING OF OUTPUT STRINGS (Specillay s for plural and case sensitivity)

-Repeat sample input cases in reverse order

-Read the problem again, specially the input and output

-Make sure you correctly initialize between testcases

-Math operations like mod, floor and ceil works differentelly on positive and negative

-Multiple edges between two nodes -Multiple spaces between input words -truncate or apporximate

-double issues, watch for -0.0 (if the double is near than zero output zero) and don't use (==, <, >) directly

-Multiple input items (same string in the input twice), use set or multiset

-Input terminating condition and output format must equal to what the problem specified

-Copy input correctly from problem statements

-watch for special cases in the input

-Integer and char overflow (multiplications & powers& Cross Products)!!

-Make use you don't use a very large infinity and add things to it which may cause overflow

-If you've a double and want to convert it to integer (multiply by 100000 or so), then add EPS first as the double 0.7 may be stored as. Watch out: "Input is a 32 integer bit" int x; cin>>x; if(x<0) x = -x; do(x); OVERFLOW: -2^31 should not be positived in int var.

0.69999999

-HashSet and HashMap don't sort, TreeSet and TreeMap do (C++ set and map are tree-based) -If the problem can be DPed then do it this way (safer than greedy)

-After all, you may have got the problem the wrong way, let a fresh member read it and hear from him (don't affect him)

-not a number(NAN) which comes from sqrt(-ve), (0/0) ,or acos(1.00000000001) or cos(-1.000000000001) for such case if the value is very close

to -1 or 1 make it 1.

- reading by scanf("%d ",x) to remove '\n' can remove leading spaces on the next line

TLE bugs: -Note that Choosing all combination of N items is of order 2^N using recursion and N\*(2^N) if using bitmasked loop -Use scanf instead of cin if u got TLE -avoid division, mod and multiplicatio operations if u got TLE -If the problem is DP, make sure you are using the smallest possible number of dimensions for the DP -incorrect input reading/termination (watch for empty lines)

Runtime bugs: -Index out of boundaries -Stack over flow -integer division by zero -Calling Integer.pareseInt with invalid string- incorrect input reading (getline)- empty lines in input

Presentataion error=Output formmat error: 1) Watch out diplayed lists 1 3 7 9   Do not display SPACE after last number(here 9)

2) Make sure from sepreating testscases. 2.1) Display blank line after each test case means there is a line between each test case even after the last test case. 2.2) Display blank line between test casse. --> Means ONLY between testcases

3) In C++ : memcpy and memset don't work normally with very large arrays

1- first hour is the hunt for ACES, don't interrupt the team members too much in this hour. if there is an interruption it should be for asking about something not for thinking with you in the idea.

2- the ACE problem is the addition, multiplication or sorting problem such that it's not harder than Div2-250 or the lines of code doesn't exceed 20 lines. **it's a must that the problem doesn't need the strategy and the problem can be solved inside the main**.

3- read the problem statement till the end, take your time to check the input and the output, and take care that the sample input and output may have the key to the problem solution.

4- make your code small, simple, smart

contest scenario: In the first hour do the following:

1- no interrupts, hunting for aces, and reading problems as much as you can.

2- read the problems to the end including the input and the output section and put a rough estimate for the problem.

3- never not to complete reading a problem to the end.

starting from second hour:

1- all problems codes must be written on papers.

2- the written code should be written in a clean way.

3- the code should be scanned from the papers to the machine and compile.

starting from the third hour:

1- the score board is a good guide to see which problems you should solve.

2- schedule for the next 2 hours which problems to start with and which to delay.

in the last hour:

1- do not start coding a problem in the last hour unless you got accepted in all the other tried problems.

2- do your best to solve all the written problems.

[«](http://wahab.homeip.net/JSPWiki/Wiki.jsp?page=Contest%20Solving%20Strategy#top)

## WHY WRONG ANSWER

-CHECK THE SPELLING OF OUTPUT STRINGS (Specially s for plural and case sensitivity)

-Repeat sample input cases in reverse order

-Compete with problem set instead of team, use score board only to know which problems are solved

-Read the problem again, specially the input and output

-Make sure you correctly initialize between test cases

-Multiple edges between two nodes

-Multiple spaces between input words

-truncate or approximate

-double issues, watch for -0.0 (if the double is near than zero output zero) and don't use (==, <, >) directly

-Multiple input items (same string in the input twice), use set or multiset

-Input terminating condition and output format must equal to what the problem specified

-Copy input correctly from problem statements

-Watch for special cases in the input

-Integer and char overflow!!

-Make sure you don't use a very large infinity and add things to it which may cause overflow (E.g. in DP)

-Small infinity may be wrong (if it smaller than what u calc)

-overflow: multiplications( cross product ) & powers & Base conversions & DP counting problems.

-Check CAREFULLY input stopping conditions. E.g. Input terminate with line START with # or CONTAINS #

-If you've a double and want to convert it to integer (multiply by 100000 or so), then add EPS first as the double 0.7 may be stored as 0.69999999

-HashSet and HashMap don't sort, TreeSet and TreeMap do (C++ set and map are tree-based)

-If the problem can be DPed then do it this way (safer than greedy)

-After all, you may have got the problem the wrong way, let a fresh member read it and hear from him (don't affect him)

-not a number(NAN) which comes from sqrt(-ve), (0/0) ,or cos(1.00000000001) or cos(-1.000000000001) for such case if the value is very close to -1 or 1 make it 1.

-make sure when u are flooring a -ve integer that u floor it to the nearest less integer, for example Floor(-2.3) = -3 but Floor(2.3)= 2.

- Other tricks:

- Word is "sequence of upper/lower case letters". then ali is 1 word, X-Ray is 2 words

- You will operate on string of letters (this do not mean Latin letters a-z, this is bigger)

- Given 2 integers i, j, find number of primes between them. input may be 10 20 OR 20 10

- Given N\*M grid, Read N lines each start with M chars. E.g. 3\*2

-- 1st line -> ab

-- 2nd line -> cdEXTRA // use to depend on read N, M, as RE may happen

-- 3rd line -> ef

- In multiset insert add new element, but delete removes ALL instances of element

-- if multiset contains (3 3 3 3 6 9) and u delete 3 -->will be (6, 9)

- Use to read input then process it, if u did not, do not BREAK wrongly while reading.

--lp(i, 5) { cin>>x; if(!valid(x)) { ok = 0; break;} --> What about output REMINDER?

- Geometry: Is polygon simple, convex, concave? Is there duplicate points? Does it matter?

- if you are using double the maximum eps you can use is 1e-11, if you need more precision you have to use long double instead.

- if the output is longlong make sure that you use cout not printf