



DU_FireBlitz Codebook

Nayeemul Islam Swad Asif Jawad Thanic Nur Samin
@DrSwad @_timelord @Kaleidoscope25

February 17, 2019

Contents

1	Numerical algorithms	1
1.1	Number theory (modular, Chinese remainder, linear Diophantine)	1
1.2	Systems of linear equations, matrix inverse, determinant	2
2	Graph algorithms	3
2.1	Bellman-Ford shortest paths with negative edge weights (C++)	3
2.2	Dijkstra and Floyd's algorithm (C++)	3
2.3	Fast Dijkstra's algorithm	3
2.4	Strongly connected components	4
2.5	Eulerian path	4
2.6	Kruskal's algorithm	4
2.7	Minimum spanning trees	5
3	Data structures	5
3.1	Suffix array	5
3.2	Binary Indexed Tree	6
3.3	Union-find set	6
3.4	Lazy segment tree	6
3.5	Lowest common ancestor	7
4	Geometry	7
4.1	Convex hull	7
4.2	Miscellaneous geometry	8
4.3	Slow Delaunay triangulation	10
5	Combinatorial optimization	10
5.1	Dense max-flow	10
5.2	Min-cost max-flow	11
6	Miscellaneous	12
6.1	Longest increasing subsequence	12
6.2	Prime numbers	12
6.3	C++ input/output	12
6.4	Knuth-Morris-Pratt	12
6.5	Topological sort (C++)	13
6.6	Random STL stuff	13
6.7	Fast exponentiation	13
6.8	Longest common subsequence	14

1 Numerical algorithms

1.1 Number theory (modular, Chinese remainder, linear Diophantine)

*// This is a collection of useful code for solving problems that
// involve modular linear equations. Note that all of the
// algorithms described here work on nonnegative integers.*

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef vector<int> VI;
typedef pair<int, int> PII;

// return a % b (positive value)
int mod(int a, int b) {
    return ((a%b) + b) % b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
    while (b) { int t = a%b; a = b; b = t; }
    return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
    return a / gcd(a, b)*b;
}

// (a^b) mod m via successive squaring
int powermod(int a, int b, int m)
{
    int ret = 1;
    while (b)
    {
        if (b & 1) ret = mod(ret*a, m);
        a = mod(a*a, m);
        b >>= 1;
    }
    return ret;
}

// returns g = gcd(a, b); finds x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
    int xx = y = 0;
    int yy = x = 1;
    while (b) {
        int q = a / b;
        int t = b; b = a%b; a = t;
        t = xx; xx = x - q*xx; x = t;
        t = yy; yy = y - q*yy; y = t;
    }
    return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
    int x, y;
    VI ret;
    int g = extended_euclid(a, n, x, y);
    if (!(b%g)) {
        x = mod(x*(b / g), n);
        for (int i = 0; i < g; i++)
            ret.push_back(mod(x + i*(n / g), n));
    }
    return ret;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse(int a, int n) {
    int x, y;
    int g = extended_euclid(a, n, x, y);
    if (g > 1) return -1;
    return mod(x, n);
}
```

```
// Chinese remainder theorem (special case): find z such that
// z % m1 = r1, z % m2 = r2. Here, z is unique modulo M = lcm(m1, m2).
// Return (z, M). On failure, M = -1.
PII chinese_remainder_theorem(int m1, int r1, int m2, int r2) {
    int s, t;
    int g = extended_euclid(m1, m2, s, t);
    if (r1 % g != r2 % g) return make_pair(0, -1);
    return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);
}

// Chinese remainder theorem: find z such that
// z % m[i] = r[i] for all i. Note that the solution is
// unique modulo M = lcm_i (m[i]). Return (z, M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
PII chinese_remainder_theorem(const VI &m, const VI &r) {
    PII ret = make_pair(r[0], m[0]);
    for (int i = 1; i < m.size(); i++) {
        ret = chinese_remainder_theorem(ret.second, ret.first, m[i], r[i]);
        if (ret.second == -1) break;
    }
    return ret;
}

// computes x and y such that ax + by = c
// returns whether the solution exists
bool linear_diophantine(int a, int b, int c, int &x, int &y) {
    if (!a && !b)
    {
        if (c) return false;
        x = 0; y = 0;
        return true;
    }
    if (!a)
    {
        if (c % b) return false;
        x = 0; y = c / b;
        return true;
    }
    if (!b)
    {
        if (c % a) return false;
        x = c / a; y = 0;
        return true;
    }
    int g = gcd(a, b);
    if (c % g) return false;
    x = c / g * mod_inverse(a / g, b / g);
    y = (c - a*x) / b;
    return true;
}

int main() {
    // expected: 2
    cout << gcd(14, 30) << endl;

    // expected: 2 -2 1
    int x, y;
    int g = extended_euclid(14, 30, x, y);
    cout << g << " " << x << " " << y << endl;

    // expected: 95 451
    VI sols = modular_linear_equation_solver(14, 30, 100);
    for (int i = 0; i < sols.size(); i++) cout << sols[i] << " ";
    cout << endl;

    // expected: 8
    cout << mod_inverse(8, 9) << endl;

    // expected: 23 105
    // 11 12
    PII ret = chinese_remainder_theorem(VI({ 3, 5, 7 }), VI({ 2, 3, 2 }));
    cout << ret.first << " " << ret.second << endl;
    ret = chinese_remainder_theorem(VI({ 4, 6 }), VI({ 3, 5 }));
    cout << ret.first << " " << ret.second << endl;

    // expected: 5 -15
    if (!linear_diophantine(7, 2, 5, x, y)) cout << "ERROR" << endl;
    cout << x << " " << y << endl;
    return 0;
}
```

1.2 Systems of linear equations, matrix inverse, determinant

```
// Gauss-Jordan elimination with full pivoting.
```

```
//
// Uses:
// (1) solving systems of linear equations (AX=B)
// (2) inverting matrices (AX=I)
// (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT: a[][] = an nxn matrix
// b[][] = an nxm matrix
//
// OUTPUT: X = an nxm matrix (stored in b[][])
// A^-1 = an nxn matrix (stored in a[][])
// returns determinant of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
    const int n = a.size();
    const int m = b[0].size();
    VI irow(n), icol(n), ipiv(n);
    T det = 1;

    for (int i = 0; i < n; i++) {
        int pj = -1, pk = -1;
        for (int j = 0; j < n; j++) if (!ipiv[j])
            for (int k = 0; k < n; k++) if (!ipiv[k])
                if (fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
        if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }
        ipiv[pj]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        if (pj != pk) det *= -1;
        irow[i] = pj;
        icol[i] = pk;

        T c = 1.0 / a[pk][pk];
        det *= a[pk][pk];
        a[pk][pk] = 1.0;
        for (int p = 0; p < n; p++) a[pk][p] *= c;
        for (int p = 0; p < m; p++) b[pk][p] *= c;
        for (int p = 0; p < n; p++) if (p != pk) {
            c = a[p][pk];
            a[p][pk] = 0;
            for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
            for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
        }

        for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
            for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
        }

        return det;
    }
}

int main() {
    const int n = 4;
    const int m = 2;
    double A[n][n] = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
    double B[n][m] = { {1,2},{4,3},{5,6},{8,7} };
    VVT a(n), b(n);
    for (int i = 0; i < n; i++) {
        a[i] = VT(A[i], A[i] + n);
        b[i] = VT(B[i], B[i] + m);
    }

    double det = GaussJordan(a, b);

    // expected: 60
    cout << "Determinant: " << det << endl;

    // expected: -0.233333 0.166667 0.133333 0.066667
    // 0.166667 0.166667 0.333333 -0.333333
    // 0.233333 0.833333 -0.133333 -0.066667
    // 0.05 -0.75 -0.1 0.2
    cout << "Inverse: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }
```

```

}

// expected: 1.63333 1.3
//           -0.166667 0.5
//           2.36667 1.7
//           -1.85 -1.35
cout << "Solution: " << endl;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++)
        cout << b[i][j] << ' ';
    cout << endl;
}
}

```

2 Graph algorithms

2.1 Bellman-Ford shortest paths with negative edge weights (C++)

```

// This function runs the Bellman-Ford algorithm for single source
// shortest paths with negative edge weights. The function returns
// false if a negative weight cycle is detected. Otherwise, the
// function returns true and dist[i] is the length of the shortest
// path from start to i.
//
// Running time: O(|V|^3)
//
// INPUT:  start, w[i][j] = cost of edge from i to j
// OUTPUT: dist[i] = min weight path from start to i
//         prev[i] = previous node on the best path from the
//               start node

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;

bool BellmanFord (const VVT &w, VT &dist, VI &prev, int start){
    int n = w.size();
    prev = VI(n, -1);
    dist = VT(n, 1000000000);
    dist[start] = 0;

    for (int k = 0; k < n; k++){
        for (int i = 0; i < n; i++){
            for (int j = 0; j < n; j++){
                if (dist[j] > dist[i] + w[i][j]){
                    if (k == n-1) return false;
                    dist[j] = dist[i] + w[i][j];
                    prev[j] = i;
                }
            }
        }
    }

    return true;
}

```

2.2 Dijkstra and Floyd's algorithm (C++)

```

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;

```

```

typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;

// This function runs Dijkstra's algorithm for single source
// shortest paths. No negative cycles allowed!
//
// Running time: O(|V|^2)
//
// INPUT:  start, w[i][j] = cost of edge from i to j
// OUTPUT: dist[i] = min weight path from start to i
//         prev[i] = previous node on the best path from the
//               start node

```

```

void Dijkstra (const VVT &w, VT &dist, VI &prev, int start){
    int n = w.size();
    VI found(n);
    prev = VI(n, -1);
    dist = VT(n, 1000000000);
    dist[start] = 0;

```

```

    while (start != -1){
        found[start] = true;
        int best = -1;
        for (int k = 0; k < n; k++) if (!found[k]){
            if (dist[k] > dist[start] + w[start][k]){
                dist[k] = dist[start] + w[start][k];
                prev[k] = start;
            }
            if (best == -1 || dist[k] < dist[best]) best = k;
        }
        start = best;
    }
}

```

```

// This function runs the Floyd-Warshall algorithm for all-pairs
// shortest paths. Also handles negative edge weights. Returns true
// if a negative weight cycle is found.
//
// Running time: O(|V|^3)
//
// INPUT:  w[i][j] = weight of edge from i to j
// OUTPUT: w[i][j] = shortest path from i to j
//         prev[i][j] = node before j on the best path starting at i

```

```

bool FloydWarshall (VVT &w, VVI &prev){
    int n = w.size();
    prev = VVI (n, VI(n, -1));

    for (int k = 0; k < n; k++){
        for (int i = 0; i < n; i++){
            for (int j = 0; j < n; j++){
                if (w[i][j] > w[i][k] + w[k][j]){
                    w[i][j] = w[i][k] + w[k][j];
                    prev[i][j] = k;
                }
            }
        }
    }

    // check for negative weight cycles
    for(int i=0;i<n;i++){
        if (w[i][i] < 0) return false;
    }
    return true;
}

```

2.3 Fast Dijkstra's algorithm

```

// Implementation of Dijkstra's algorithm using adjacency lists
// and priority queue for efficiency.
//
// Running time: O(|E| log |V|)

```

```

#include <queue>
#include <cstdio>

using namespace std;
const int INF = 2000000000;
typedef pair<int, int> PII;

int main() {
    int N, s, t;
    scanf("%d%d%d", &N, &s, &t);
    vector<vector<PII>> edges(N);
    for (int i = 0; i < N; i++) {

```

```

int M;
scanf("%d", &M);
for (int j = 0; j < M; j++) {
    int vertex, dist;
    scanf("%d%d", &vertex, &dist);
    edges[i].push_back(make_pair(dist, vertex)); // note order of arguments here
}

// use priority queue in which top element has the "smallest" priority
priority_queue<PII, vector<PII>, greater<PII> > Q;
vector<int> dist(N, INF), dad(N, -1);
Q.push(make_pair(0, s));
dist[s] = 0;
while (!Q.empty()) {
    PII p = Q.top();
    Q.pop();
    int here = p.second;
    if (here == t) break;
    if (dist[here] != p.first) continue;

    for (vector<PII>::iterator it = edges[here].begin(); it != edges[here].end(); it++) {
        if (dist[here] + it->first < dist[it->second]) {
            dist[it->second] = dist[here] + it->first;
            dad[it->second] = here;
            Q.push(make_pair(dist[it->second], it->second));
        }
    }

    printf("%d\n", dist[t]);
    if (dist[t] < INF)
        for (int i = t; i != -1; i = dad[i])
            printf("%d%c", i, (i == s ? '\n' : ' '));

    return 0;
}

/*
Sample input:
5 0 4
2 1 2 3 1
2 2 4 4 5
3 1 4 3 3 4 1
2 0 1 2 3
2 1 5 2 1

Expected:
5
4 2 3 0
*/

```

2.4 Strongly connected components

```

#include<memory.h>
struct edge{int e, nxt;};
int V, E;
edge e[MAXE], er[MAXE];
int sp[MAXV], spr[MAXV];
int group_cnt, group_num[MAXV];
bool v[MAXV];
int stk[MAXV];
void fill_forward(int x)
{
    int i;
    v[x]=true;
    for(i=sp[x];i;i=e[i].nxt) if(!v[e[i].e]) fill_forward(e[i].e);
    stk[++stk[0]]=x;
}
void fill_backward(int x)
{
    int i;
    v[x]=false;
    group_num[x]=group_cnt;
    for(i=spr[x];i;i=er[i].nxt) if(v[er[i].e]) fill_backward(er[i].e);
}
void add_edge(int v1, int v2) //add edge v1->v2
{
    e[++E].e=v2; e[E].nxt=sp[v1]; sp[v1]=E;
    er[E].e=v1; er[E].nxt=spr[v2]; spr[v2]=E;
}
void SCC()
{
    int i;
    stk[0]=0;
    memset(v, false, sizeof(v));
    for(i=1;i<=V;i++) if(!v[i]) fill_forward(i);
    group_cnt=0;
}

```

```

for(i=stk[0];i>=1;i--) if(v[stk[i]]){group_cnt++; fill_backward(stk[i]);}
}

```

2.5 Eulerian path

```

struct Edge;
typedef list<Edge>::iterator iter;

struct Edge
{
    int next_vertex;
    iter reverse_edge;

    Edge(int next_vertex)
        :next_vertex(next_vertex)
    { }
};

const int max_vertices = ;
int num_vertices;
list<Edge> adj[max_vertices]; // adjacency list

vector<int> path;

void find_path(int v)
{
    while(adj[v].size() > 0)
    {
        int vn = adj[v].front().next_vertex;
        adj[vn].erase(adj[v].front().reverse_edge);
        adj[v].pop_front();
        find_path(vn);
    }
    path.push_back(v);
}

void add_edge(int a, int b)
{
    adj[a].push_front(Edge(b));
    iter ita = adj[a].begin();
    adj[b].push_front(Edge(a));
    iter itb = adj[b].begin();
    ita->reverse_edge = itb;
    itb->reverse_edge = ita;
}

```

2.6 Kruskal's algorithm

```

/*
Uses Kruskal's Algorithm to calculate the weight of the minimum spanning
forest (union of minimum spanning trees of each connected component) of
a possibly disjoint graph, given in the form of a matrix of edge weights
(-1 if no edge exists). Returns the weight of the minimum spanning
forest (also calculates the actual edges - stored in T). Note: uses a
disjoint-set data structure with amortized (effectively) constant time per
union/find. Runs in O(E*log(E)) time.
*/

#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

typedef int T;

struct edge
{
    int u, v;
    T d;
};

struct edgeCmp
{
    int operator()(const edge& a, const edge& b) { return a.d > b.d; }
};

int find(vector<int>& C, int x) { return (C[x] == x) ? x : C[x] = find(C, C[x]); }

T Kruskal(vector<vector<T>>& w)
{
    int n = w.size();
}

```

```

T weight = 0;

vector<int> C(n), R(n);
for(int i=0; i<n; i++) { C[i] = i; R[i] = 0; }

vector<edge> T;
priority_queue<edge, vector<edge>, edgeCmp> E;

for(int i=0; i<n; i++)
    for(int j=i+1; j<n; j++)
        if(w[i][j] >= 0)
        {
            edge e;
            e.u = i; e.v = j; e.d = w[i][j];
            E.push(e);
        }

while(T.size() < n-1 && !E.empty())
{
    edge cur = E.top(); E.pop();

    int uc = find(C, cur.u), vc = find(C, cur.v);
    if(uc != vc)
    {
        T.push_back(cur); weight += cur.d;

        if(R[uc] > R[vc]) C[vc] = uc;
        else if(R[vc] > R[uc]) C[uc] = vc;
        else { C[vc] = uc; R[uc]++; }
    }
}

return weight;
}

int main()
{
    int wa[6][6] = {
        { 0, -1, 2, -1, 7, -1 },
        { -1, 0, -1, 2, -1, -1 },
        { 2, -1, 0, -1, 8, 6 },
        { -1, 2, -1, 0, -1, -1 },
        { 7, -1, 8, -1, 0, 4 },
        { -1, -1, 6, -1, 4, 0 } };

    vector<vector<int>> w(6, vector<int>(6));

    for(int i=0; i<6; i++)
        for(int j=0; j<6; j++)
            w[i][j] = wa[i][j];

    cout << Kruskal(w) << endl;
    cin >> wa[0][0];
}

```

2.7 Minimum spanning trees

```

// This function runs Prim's algorithm for constructing minimum
// weight spanning trees.
//
// Running time: O(|V|^2)
//
// INPUT:   w[i][j] = cost of edge from i to j
//
// NOTE: Make sure that w[i][j] is nonnegative and
//        symmetric. Missing edges should be given -1
//        weight.
//
// OUTPUT:  edges = list of pair<int,int> in minimum spanning tree
//          return total weight of tree

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

```

```

T Prim (const VVT &w, VPII &edges){
    int n = w.size();
    VI found (n);
    VI prev (n, -1);
    VT dist (n, 1000000000);
    int here = 0;
    dist[here] = 0;

    while (here != -1){
        found[here] = true;
        int best = -1;
        for (int k = 0; k < n; k++) if (!found[k]){
            if (w[here][k] != -1 && dist[k] > w[here][k]){
                dist[k] = w[here][k];
                prev[k] = here;
            }
            if (best == -1 || dist[k] < dist[best]) best = k;
        }
        here = best;
    }

    T tot_weight = 0;
    for (int i = 0; i < n; i++) if (prev[i] != -1){
        edges.push_back (make_pair (prev[i], i));
        tot_weight += w[prev[i]][i];
    }
    return tot_weight;
}

int main(){
    int ww[5][5] = {
        {0, 400, 400, 300, 600},
        {400, 0, 3, -1, 7},
        {400, 3, 0, 2, 0},
        {300, -1, 2, 0, 5},
        {600, 7, 0, 5, 0}
    };
    VVT w(5, VT(5));
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            w[i][j] = ww[i][j];

    // expected: 305
    // 2 1
    // 3 2
    // 0 3
    // 2 4

    VPII edges;
    cout << Prim (w, edges) << endl;
    for (int i = 0; i < edges.size(); i++)
        cout << edges[i].first << " " << edges[i].second << endl;
}

```

3 Data structures

3.1 Suffix array

```

// Suffix array construction in O(L log^2 L) time. Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:   string s
//
// OUTPUT:  array suffix[] such that suffix[i] = index (from 0 to L-1)
//          of substring s[i..L-1] in the list of sorted suffixes.
//          That is, if we take the inverse of the permutation suffix[],
//          we get the actual suffix array.

#include <vector>
#include <iostream>
#include <string>

using namespace std;

struct SuffixArray {
    const int L;
    string s;
    vector<vector<int>> > P;
    vector<pair<pair<int,int>,int>> > M;

    SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L) {
        for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
        for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
            P.push_back(vector<int>(L, 0));

```

```

    for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);
    sort(M.begin(), M.end());
    for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;
}

vector<int> GetSuffixArray() { return P.back(); }

// returns the length of the longest common prefix of s[i...L-1] and s[j...L-1]
int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
        if (P[k][i] == P[k][j]) {
            i += 1 << k;
            j += 1 << k;
            len += 1 << k;
        }
    }
    return len;
}

// BEGIN CUT
// The following code solves UVA problem 11512: GATTACA.
#define TESTING
#ifdef TESTING
int main() {
    int T;
    cin >> T;
    for (int caseno = 0; caseno < T; caseno++) {
        string s;
        cin >> s;
        SuffixArray array(s);
        vector<int> v = array.GetSuffixArray();
        int bestlen = -1, bestpos = -1, bestcount = 0;
        for (int i = 0; i < s.length(); i++) {
            int len = 0, count = 0;
            for (int j = i+1; j < s.length(); j++) {
                int l = array.LongestCommonPrefix(i, j);
                if (l >= len) {
                    if (l > len) count = 2; else count++;
                    len = l;
                }
            }
            if (len > bestlen || len == bestlen && s.substr(bestpos, bestlen) > s.substr(i, len)) {
                bestlen = len;
                bestcount = count;
                bestpos = i;
            }
        }
        if (bestlen == 0) {
            cout << "No repetitions found!" << endl;
        } else {
            cout << s.substr(bestpos, bestlen) << " " << bestcount << endl;
        }
    }
}

#else
// END CUT
int main() {

    // bobocel is the 0'th suffix
    // obocel is the 5'th suffix
    // bocel is the 1'st suffix
    // ocel is the 6'th suffix
    // cel is the 2'nd suffix
    // el is the 3'rd suffix
    // l is the 4'th suffix
    SuffixArray suffix("bobocel");
    vector<int> v = suffix.GetSuffixArray();

    // Expected output: 0 5 1 6 2 3 4
    //
    for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    cout << suffix.LongestCommonPrefix(0, 2) << endl;
}

// BEGIN CUT
#endif
// END CUT

```

3.2 Binary Indexed Tree

```
#include <iostream>
```

```

using namespace std;

#define LOGSZ 17

int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);

// add v to value at x
void set(int x, int v) {
    while(x <= N) {
        tree[x] += v;
        x += (x & -x);
    }
}

// get cumulative sum up to and including x
int get(int x) {
    int res = 0;
    while(x) {
        res += tree[x];
        x -= (x & -x);
    }
    return res;
}

// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
    int idx = 0, mask = N;
    while(mask && idx < N) {
        int t = idx + mask;
        if(x >= tree[t]) {
            idx = t;
            x -= tree[t];
        }
        mask >>= 1;
    }
    return idx;
}

```

3.3 Union-find set

```

#include <iostream>
#include <vector>
using namespace std;
struct UnionFind {
    vector<int> C;
    UnionFind(int n) : C(n) { for (int i = 0; i < n; i++) C[i] = i; }
    int find(int x) { return (C[x] == x) ? x : C[x] = find(C[x]); }
    void merge(int x, int y) { C[find(x)] = find(y); }
};

int main()
{
    int n = 5;
    UnionFind uf(n);
    uf.merge(0, 2);
    uf.merge(1, 0);
    uf.merge(3, 4);
    for (int i = 0; i < n; i++) cout << i << " " << uf.find(i) << endl;
    return 0;
}

```

3.4 Lazy segment tree

```

public class SegmentTreeRangeUpdate {
    public long[] leaf;
    public long[] update;
    public int origSize;
    public SegmentTreeRangeUpdate(int[] list) {
        origSize = list.length;
        leaf = new long[4*list.length];
        update = new long[4*list.length];
        build(1,0,list.length-1,list);
    }

    public void build(int curr, int begin, int end, int[] list) {
        if(begin == end)
            leaf[curr] = list[begin];
        else {
            int mid = (begin+end)/2;
            build(2 + curr, begin, mid, list);
            build(2 + curr + 1, mid+1, end, list);
            leaf[curr] = leaf[2+curr] + leaf[2+curr+1];
        }
    }
}

```

```

public void update(int begin, int end, int val) {
    update(1, 0, origSize-1, begin, end, val);
}
public void update(int curr, int tBegin, int tEnd, int begin, int end, int val) {
    if(tBegin >= begin && tEnd <= end)
        update[curr] += val;
    else {
        leaf[curr] += (Math.min(end, tEnd) - Math.max(begin, tBegin) + 1) * val;
        int mid = (tBegin + tEnd) / 2;
        if(mid >= begin && tBegin <= end)
            update(2*curr, tBegin, mid, begin, end, val);
        if(tEnd >= begin && mid+1 <= end)
            update(2*curr+1, mid+1, tEnd, begin, end, val);
    }
}
public long query(int begin, int end) {
    return query(1, 0, origSize-1, begin, end);
}
public long query(int curr, int tBegin, int tEnd, int begin, int end) {
    if(tBegin >= begin && tEnd <= end) {
        if(update[curr] != 0) {
            leaf[curr] += (tEnd - tBegin + 1) * update[curr];
            if(2*curr < update.length) {
                update[2*curr] += update[curr];
                update[2*curr+1] += update[curr];
            }
            update[curr] = 0;
        }
        return leaf[curr];
    }
    else {
        leaf[curr] += (tEnd - tBegin + 1) * update[curr];
        if(2*curr < update.length) {
            update[2*curr] += update[curr];
            update[2*curr+1] += update[curr];
        }
        update[curr] = 0;
        int mid = (tBegin + tEnd) / 2;
        long ret = 0;
        if(mid >= begin && tBegin <= end)
            ret += query(2*curr, tBegin, mid, begin, end);
        if(tEnd >= begin && mid+1 <= end)
            ret += query(2*curr+1, mid+1, tEnd, begin, end);
        return ret;
    }
}
}
}

```

```

p = A[p][i];

if(p == q)
    return p;

// "binary search" for the LCA
for(int i = log_num_nodes; i >= 0; i--)
    if(A[p][i] != -1 && A[p][i] != A[q][i])
    {
        p = A[p][i];
        q = A[q][i];
    }

return A[p][0];
}

int main(int argc, char* argv[])
{
    // read num_nodes, the total number of nodes
    log_num_nodes = lb(num_nodes);

    for(int i = 0; i < num_nodes; i++)
    {
        int p;
        // read p, the parent of node i or -1 if node i is the root
        A[i][0] = p;
        if(p != -1)
            children[p].push_back(i);
        else
            root = i;
    }

    // precompute A using dynamic programming
    for(int j = 1; j <= log_num_nodes; j++)
        for(int i = 0; i < num_nodes; i++)
            if(A[i][j-1] != -1)
                A[i][j] = A[A[i][j-1]][j-1];
            else
                A[i][j] = -1;

    // precompute L
    DFS(root, 0);

    return 0;
}

```

3.5 Lowest common ancestor

```

const int max_nodes, log_max_nodes;
int num_nodes, log_num_nodes, root;

vector<int> children[max_nodes]; // children[i] contains the children of node i
int A[max_nodes][log_max_nodes+1]; // A[i][j] is the 2^j-th ancestor of node i, or -1 if that
// ancestor does not exist
int L[max_nodes]; // L[i] is the distance between node i and the root

// floor of the binary logarithm of n
int lb(unsigned int n)
{
    if(n==0)
        return -1;
    int p = 0;
    if (n >= 1<<16) { n >>= 16; p += 16; }
    if (n >= 1<< 8) { n >>= 8; p += 8; }
    if (n >= 1<< 4) { n >>= 4; p += 4; }
    if (n >= 1<< 2) { n >>= 2; p += 2; }
    if (n >= 1<< 1) { p += 1; }
    return p;
}

void DFS(int i, int l)
{
    L[i] = l;
    for(int j = 0; j < children[i].size(); j++)
        DFS(children[i][j], l+1);
}

int LCA(int p, int q)
{
    // ensure node p is at least as deep as node q
    if(L[p] < L[q])
        swap(p, q);

    // "binary search" for the ancestor of node p situated on the same level as q
    for(int i = log_num_nodes; i >= 0; i--)
        if(L[p] - (1<<i) >= L[q])

```

4 Geometry

4.1 Convex hull

```

// Compute the 2D convex hull of a set of points using the monotone chain
// algorithm. Eliminate redundant points from the hull if REMOVE_REDUNDANT is
// #defined.
//
// Running time: O(n log n)
//
// INPUT: a vector of input points, unordered.
// OUTPUT: a vector of points in the convex hull, counterclockwise, starting
// with bottommost/leftmost point

#include <stdio>
#include <cassert>
#include <vector>
#include <algorithm>
#include <cmath>
// BEGIN CUT
#include <map>
// END CUT

using namespace std;

#define REMOVE_REDUNDANT

typedef double T;
const T EPS = 1e-7;
struct PT {
    T x, y;
    PT() {}
    PT(T x, T y) : x(x), y(y) {}
    bool operator<(const PT &rhs) const { return make_pair(y,x) < make_pair(rhs.y,rhs.x); }
    bool operator==(const PT &rhs) const { return make_pair(y,x) == make_pair(rhs.y,rhs.x); }
};

```

```

T cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
T area2(PT a, PT b, PT c) { return cross(a,b) + cross(b,c) + cross(c,a); }

#ifdef REMOVE_REDUNDANT
bool between(const PT &a, const PT &b, const PT &c) {
    return (fabs(area2(a,b,c)) < EPS && (a.x-b.x)*(c.x-b.x) <= 0 && (a.y-b.y)*(c.y-b.y) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
    sort(pts.begin(), pts.end());
    pts.erase(unique(pts.begin(), pts.end()), pts.end());
    vector<PT> up, dn;
    for (int i = 0; i < pts.size(); i++) {
        while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
        while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
        up.push_back(pts[i]);
        dn.push_back(pts[i]);
    }
    pts = dn;
    for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
    if (pts.size() <= 2) return;
    dn.clear();
    dn.push_back(pts[0]);
    dn.push_back(pts[1]);
    for (int i = 2; i < pts.size(); i++) {
        if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
        dn.push_back(pts[i]);
    }
    if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
        dn[0] = dn.back();
        dn.pop_back();
    }
    pts = dn;
#endif
}

// BEGIN CUT
// The following code solves SPOJ problem #26: Build the Fence (BSHEEP)

int main() {
    int t;
    scanf("%d", &t);
    for (int caseno = 0; caseno < t; caseno++) {
        int n;
        scanf("%d", &n);
        vector<PT> v(n);
        for (int i = 0; i < n; i++) scanf("%lf%lf", &v[i].x, &v[i].y);
        vector<PT> h(v);
        map<PT,int> index;
        for (int i = n-1; i >= 0; i--) index[v[i]] = i+1;
        ConvexHull(h);

        double len = 0;
        for (int i = 0; i < h.size(); i++) {
            double dx = h[i].x - h[(i+1)%h.size()].x;
            double dy = h[i].y - h[(i+1)%h.size()].y;
            len += sqrt(dx*dx+dy*dy);
        }

        if (caseno > 0) printf("\n");
        printf("%.2f\n", len);
        for (int i = 0; i < h.size(); i++) {
            if (i > 0) printf(" ");
            printf("%d", index[h[i]]);
        }
        printf("\n");
    }
}

// END CUT

```

4.2 Miscellaneous geometry

// C++ routines for computational geometry.

```

#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

using namespace std;

double INF = 1e100;
double EPS = 1e-12;

```

```

struct PT {
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c); }
    PT operator / (double c) const { return PT(x/c, y/c); }
};

double dot(PT p, PT q) { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q,p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
    return os << "(" << p.x << ", " << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y,p.x); }
PT RotateCW90(PT p) { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
    return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
    return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
    double r = dot(b-a,b-a);
    if (fabs(r) < EPS) return a;
    r = dot(c-a, b-a)/r;
    if (r < 0) return a;
    if (r > 1) return b;
    return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
    double a, double b, double c, double d)
{
    return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
    return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a, b, c, d)
        && fabs(cross(a-b, a-c)) < EPS
        && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if (LinesCollinear(a, b, c, d)) {
        if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
            dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
        if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
            return false;
        return true;
    }
    if (cross(d-a, b-a) + cross(c-a, b-a) > 0) return false;
    if (cross(a-c, d-c) + cross(b-c, d-c) > 0) return false;
    return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
    b=b-a; d=d-c; c=c-a;
    assert(dot(b, b) > EPS && dot(d, d) > EPS);
    return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {

```



```

b=(a+b)/2;
c=(a+c)/2;
return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
    bool c = 0;
    for (int i = 0; i < p.size(); i++){
        int j = (i+1)%p.size();
        if ((p[i].y <= q.y && q.y < p[j].y ||
            p[j].y <= q.y && q.y < p[i].y) &&
            q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
            c = !c;
    }
    return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
    for (int i = 0; i < p.size(); i++)
        if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
            return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
    vector<PT> ret;
    b = b-a;
    a = a-c;
    double A = dot(b, b);
    double B = dot(a, b);
    double C = dot(a, a) - r*r;
    double D = B*B - A*C;
    if (D < -EPS) return ret;
    ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
    if (D > EPS)
        ret.push_back(c+a+b*(-B-sqrt(D))/A);
    return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
    vector<PT> ret;
    double d = sqrt(dist2(a, b));
    if (d > r+R || d+min(r, R) < max(r, R)) return ret;
    double x = (d*d-R*R+r*r)/(2*d);
    double y = sqrt(r*r-x*x);
    PT v = (b-a)/d;
    ret.push_back(a+v*x + RotateCCW90(v)*y);
    if (y > 0)
        ret.push_back(a+v*x - RotateCCW90(v)*y);
    return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion. Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
    double area = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        area += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
    return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
    PT c(0,0);
    double scale = 6.0 + ComputeSignedArea(p);
    for (int i = 0; i < p.size(); i++){
        int j = (i+1) % p.size();
        c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

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

```

```

bool IsSimple(const vector<PT> &p) {
    for (int i = 0; i < p.size(); i++) {
        for (int k = i+1; k < p.size(); k++) {
            int j = (i+1) % p.size();
            int l = (k+1) % p.size();
            if (i == 1 || j == k) continue;
            if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

int main() {
    // expected: (-5,2)
    cerr << RotateCCW90(PT(2,5)) << endl;

    // expected: (5,-2)
    cerr << RotateCW90(PT(2,5)) << endl;

    // expected: (-5,2)
    cerr << RotateCCW(PT(2,5), M_PI/2) << endl;

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

    // expected: (5,2) (7.5,3) (2.5,1)
    cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
        << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
        << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

    // expected: 6.78903
    cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

    // expected: 1 0 1
    cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
        << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
        << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

    // expected: 0 0 1
    cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
        << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
        << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

    // expected: 1 1 1 0
    cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
        << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

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

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

    vector<PT> v;
    v.push_back(PT(0,0));
    v.push_back(PT(5,0));
    v.push_back(PT(5,5));
    v.push_back(PT(0,5));

    // expected: 1 1 1 0 0
    cerr << PointInPolygon(v, PT(2,2)) << " "
        << PointInPolygon(v, PT(2,0)) << " "
        << PointInPolygon(v, PT(0,2)) << " "
        << PointInPolygon(v, PT(5,2)) << " "
        << PointInPolygon(v, PT(2,5)) << endl;

    // expected: 0 1 1 1 1
    cerr << PointOnPolygon(v, PT(2,2)) << " "
        << PointOnPolygon(v, PT(2,0)) << " "
        << PointOnPolygon(v, PT(0,2)) << " "
        << PointOnPolygon(v, PT(5,2)) << " "
        << PointOnPolygon(v, PT(2,5)) << endl;

    // expected: (1,6)
    // (5,4) (4,5)
    // blank line
    // (4,5) (5,4)
    // blank line
    // (4,5) (5,4)
    vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);

```

```

for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

// area should be 5.0
// centroid should be (1.1666666, 1.1666666)
PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
vector<PT> p(pa, pa+4);
PT c = ComputeCentroid(p);
cerr << "Area: " << ComputeArea(p) << endl;
cerr << "Centroid: " << c << endl;

return 0;
}

```

4.3 Slow Delaunay triangulation

```

// Slow but simple Delaunay triangulation. Does not handle
// degenerate cases (from O'Rourke, Computational Geometry in C)
//
// Running time: O(n^4)
//
// INPUT:  x[] = x-coordinates
//         y[] = y-coordinates
//
// OUTPUT: triples = a vector containing m triples of indices
//               corresponding to triangle vertices

#include<vector>
using namespace std;

typedef double T;

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

vector<triple> delaunayTriangulation(vector<T>& x, vector<T>& y) {
    int n = x.size();
    vector<T> z(n);
    vector<triple> ret;

    for (int i = 0; i < n; i++)
        z[i] = x[i] * x[i] + y[i] * y[i];

    for (int i = 0; i < n-2; i++) {
        for (int j = i+1; j < n; j++) {
            for (int k = i+1; k < n; k++) {
                if (j == k) continue;
                double xn = (y[j]-y[i])*(z[k]-z[i]) - (y[k]-y[i])*(z[j]-z[i]);
                double yn = (x[k]-x[i])*(z[j]-z[i]) - (x[j]-x[i])*(z[k]-z[i]);
                double zn = (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i]);
                bool flag = zn < 0;
                for (int m = 0; flag && m < n; m++)
                    flag = flag && ((x[m]-x[i])*xn +
                                     (y[m]-y[i])*yn +
                                     (z[m]-z[i])*zn <= 0);
                if (flag) ret.push_back(triple(i, j, k));
            }
        }
    }
    return ret;
}

int main()
{
    T xs[]={0, 0, 1, 0.9};
    T ys[]={0, 1, 0, 0.9};
    vector<T> x(&xs[0], &xs[4]), y(&ys[0], &ys[4]);
    vector<triple> tri = delaunayTriangulation(x, y);

    //expected: 0 1 3
    //          0 3 2

    int i;
    for(i = 0; i < tri.size(); i++)
        printf("%d %d %d\n", tri[i].i, tri[i].j, tri[i].k);
    return 0;
}

```

5 Combinatorial optimization

5.1 Dense max-flow

```

// Adjacency matrix implementation of Dinic's blocking flow algorithm.
//
// Running time:
// O(|V|^4)
//
// INPUT:
// - graph, constructed using AddEdge()
// - source
// - sink
//
// OUTPUT:
// - maximum flow value
// - To obtain the actual flow, look at positive values only.

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;

struct MaxFlow {
    int N;
    VVI cap, flow;
    VI dad, Q;

    MaxFlow(int N) :
        N(N), cap(N, VI(N)), flow(N, VI(N)), dad(N), Q(N) {}

    void AddEdge(int from, int to, int cap) {
        this->cap[from][to] += cap;
    }

    int BlockingFlow(int s, int t) {
        fill(dad.begin(), dad.end(), -1);
        dad[s] = -2;

        int head = 0, tail = 0;
        Q[tail++] = s;
        while (head < tail) {
            int x = Q[head++];
            for (int i = 0; i < N; i++) {
                if (dad[i] == -1 && cap[x][i] - flow[x][i] > 0) {
                    dad[i] = x;
                    Q[tail++] = i;
                }
            }
        }

        if (dad[t] == -1) return 0;

        int totflow = 0;
        for (int i = 0; i < N; i++) {
            if (dad[i] == -1) continue;
            int amt = cap[i][t] - flow[i][t];
            for (int j = i; amt && j != s; j = dad[j])
                amt = min(amt, cap[dad[j]][j] - flow[dad[j]][j]);
            if (amt == 0) continue;
            flow[i][t] += amt;
            flow[t][i] -= amt;
            for (int j = i; j != s; j = dad[j]) {
                flow[dad[j]][j] += amt;
                flow[j][dad[j]] -= amt;
            }
            totflow += amt;
        }

        return totflow;
    }

    int GetMaxFlow(int source, int sink) {
        int totflow = 0;
        while (int flow = BlockingFlow(source, sink))
            totflow += flow;
        return totflow;
    }
};

```

```

int main() {
    MaxFlow mf(5);
    mf.AddEdge(0, 1, 3);
    mf.AddEdge(0, 2, 4);
    mf.AddEdge(0, 3, 5);
    mf.AddEdge(0, 4, 5);
    mf.AddEdge(1, 2, 2);
    mf.AddEdge(2, 3, 4);
    mf.AddEdge(2, 4, 1);
    mf.AddEdge(3, 4, 10);

    // should print out "15"
    cout << mf.GetMaxFlow(0, 4) << endl;
}

// BEGIN CUT
// The following code solves SPOJ problem #203: Potholers (POTHOLE)

#ifdef COMMENT
int main() {
    int t;
    cin >> t;
    for (int i = 0; i < t; i++) {
        int n;
        cin >> n;
        MaxFlow mf(n);
        for (int j = 0; j < n-1; j++) {
            int m;
            cin >> m;
            for (int k = 0; k < m; k++) {
                int p;
                cin >> p;
                p--;
                int cap = (j == 0 || p == n-1) ? 1 : INF;
                mf.AddEdge(j, p, cap);
            }
        }

        cout << mf.GetMaxFlow(0, n-1) << endl;
    }
    return 0;
}
#endif

// END CUT

```

5.2 Min-cost max-flow

```

// Implementation of min cost max flow algorithm using adjacency
// matrix (Edmonds and Karp 1972). This implementation keeps track of
// forward and reverse edges separately (so you can set cap[i][j] !=
// cap[j][i]). For a regular max flow, set all edge costs to 0.
//
// Running time,  $O(|V|^2)$  cost per augmentation
// max flow:  $O(|V|^3)$  augmentations
// min cost max flow:  $O(|V|^4 * \text{MAX\_EDGE\_COST})$  augmentations
//
// INPUT:
// - graph, constructed using AddEdge()
// - source
// - sink
//
// OUTPUT:
// - (maximum flow value, minimum cost value)
// - To obtain the actual flow, look at positive values only.

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;
typedef long long L;
typedef vector<L> VL;
typedef vector<VL> VVL;
typedef pair<int, int> PII;
typedef vector<PII> VPPII;

const L INF = numeric_limits<L>::max() / 4;

struct MinCostMaxFlow {
    int N;
    VVL cap, flow, cost;
    VI found;
    VL dist, pi, width;

```

```

    VPPII dad;

    MinCostMaxFlow(int N) :
        N(N), cap(N, VL(N)), flow(N, VL(N)), cost(N, VL(N)),
        found(N, dist(N), pi(N), width(N), dad(N)) {}

    void AddEdge(int from, int to, L cap, L cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
    }

    void Relax(int s, int k, L cap, L cost, int dir) {
        L val = dist[s] + pi[s] - pi[k] + cost;
        if (cap && val < dist[k]) {
            dist[k] = val;
            dad[k] = make_pair(s, dir);
            width[k] = min(cap, width[s]);
        }
    }

    L Dijkstra(int s, int t) {
        fill(found.begin(), found.end(), false);
        fill(dist.begin(), dist.end(), INF);
        fill(width.begin(), width.end(), 0);
        dist[s] = 0;
        width[s] = INF;

        while (s != -1) {
            int best = -1;
            found[s] = true;
            for (int k = 0; k < N; k++) {
                if (found[k]) continue;
                Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
                Relax(s, k, flow[k][s], -cost[k][s], -1);
                if (best == -1 || dist[k] < dist[best]) best = k;
            }
            s = best;
        }

        for (int k = 0; k < N; k++)
            pi[k] = min(pi[k] + dist[k], INF);
        return width[t];
    }

    pair<L, L> GetMaxFlow(int s, int t) {
        L totflow = 0, totcost = 0;
        while (L amt = Dijkstra(s, t)) {
            totflow += amt;
            for (int x = t; x != s; x = dad[x].first) {
                if (dad[x].second == 1) {
                    flow[dad[x].first][x] += amt;
                    totcost += amt * cost[dad[x].first][x];
                } else {
                    flow[x][dad[x].first] -= amt;
                    totcost -= amt * cost[x][dad[x].first];
                }
            }
            return make_pair(totflow, totcost);
        }
    };

    // BEGIN CUT
    // The following code solves UVA problem #10594: Data Flow

    int main() {
        int N, M;

        while (scanf("%d%d", &N, &M) == 2) {
            VVL v(M, VL(3));
            for (int i = 0; i < M; i++)
                scanf("%Ld%Ld%Ld", &v[i][0], &v[i][1], &v[i][2]);
            L D, K;
            scanf("%Ld%Ld", &D, &K);

            MinCostMaxFlow mcmf(N+1);
            for (int i = 0; i < M; i++) {
                mcmf.AddEdge(int(v[i][0]), int(v[i][1]), K, v[i][2]);
                mcmf.AddEdge(int(v[i][1]), int(v[i][0]), K, v[i][2]);
            }
            mcmf.AddEdge(0, 1, D, 0);

            pair<L, L> res = mcmf.GetMaxFlow(0, N);

            if (res.first == D) {
                printf("%Ld\n", res.second);
            } else {
                printf("Impossible.\n");
            }
        }

        return 0;
    }

```

```
}
// END CUT
```

6 Miscellaneous

6.1 Longest increasing subsequence

```
// Given a list of numbers of length n, this routine extracts a
// longest increasing subsequence.
//
// Running time: O(n log n)
//
// INPUT: a vector of integers
// OUTPUT: a vector containing the longest increasing subsequence

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef vector<int> VI;
typedef pair<int,int> PII;
typedef vector<PII> VPII;

#define STRICTLY_INCREASNG

VI LongestIncreasingSubsequence(VI v) {
    VPII best;
    VI dad(v.size(), -1);

    for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
        PII item = make_pair(v[i], 0);
        VPII::iterator it = lower_bound(best.begin(), best.end(), item);
        item.second = i;
#else
        PII item = make_pair(v[i], i);
        VPII::iterator it = upper_bound(best.begin(), best.end(), item);
#endif
        if (it == best.end()) {
            dad[i] = (best.size() == 0 ? -1 : best.back().second);
            best.push_back(item);
        } else {
            dad[i] = it == best.begin() ? -1 : prev(it)->second;
            *it = item;
        }
    }

    VI ret;
    for (int i = best.back().second; i >= 0; i = dad[i])
        ret.push_back(v[i]);
    reverse(ret.begin(), ret.end());
    return ret;
}
```

6.2 Prime numbers

```
// O(sqrt(x)) Exhaustive Primality Test
#include <cmath>
#define EPS 1e-7
typedef long long LL;
bool IsPrimeSlow (LL x)
{
    if (x<=1) return false;
    if (x<=3) return true;
    if (!(x%2) || !(x%3)) return false;
    LL s=(LL) (sqrt((double) (x))+EPS);
    for (LL i=5; i<=s; i+=6)
    {
        if (!(x%i) || !(x%(i+2))) return false;
    }
    return true;
}

// Primes less than 1000:
//      2      3      5      7      11     13     17     19     23     29     31     37
//      41     43     47     53     59     61     67     71     73     79     83     89
//      97     101    103    107    109    113    127    131    137    139    149    151
//      157    163    167    173    179    181    191    193    197    199    211    223
```

```
//      227    229    233    239    241    251    257    263    269    271    277    281
//      283    293    307    311    313    317    331    337    347    349    353    359
//      367    373    379    383    389    397    401    409    419    421    431    433
//      439    443    449    457    461    463    467    479    487    491    499    503
//      509    521    523    541    547    557    563    569    571    577    587    593
//      599    601    607    613    617    619    631    641    643    647    653    659
//      661    673    677    683    691    701    709    719    727    733    739    743
//      751    757    761    769    773    787    797    809    811    821    823    827
//      829    839    853    857    859    863    877    881    883    887    907    911
//      919    929    937    941    947    953    967    971    977    983    991    997

// Other primes:
// The largest prime smaller than 10 is 7.
// The largest prime smaller than 100 is 97.
// The largest prime smaller than 1000 is 997.
// The largest prime smaller than 10000 is 9973.
// The largest prime smaller than 100000 is 99991.
// The largest prime smaller than 1000000 is 999983.
// The largest prime smaller than 10000000 is 9999991.
// The largest prime smaller than 100000000 is 99999989.
// The largest prime smaller than 1000000000 is 999999937.
// The largest prime smaller than 10000000000 is 9999999977.
// The largest prime smaller than 100000000000 is 9999999997.
// The largest prime smaller than 1000000000000 is 999999999973.
// The largest prime smaller than 10000000000000 is 9999999999989.
// The largest prime smaller than 100000000000000 is 99999999999937.
// The largest prime smaller than 1000000000000000 is 99999999999997.
// The largest prime smaller than 10000000000000000 is 999999999999989.
```

6.3 C++ input/output

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    // Ouput a specific number of digits past the decimal point,
    // in this case 5
    cout.setf(ios::fixed); cout << setprecision(5);
    cout << 100.0/7.0 << endl;
    cout.unsetf(ios::fixed);

    // Output the decimal point and trailing zeros
    cout.setf(ios::showpoint);
    cout << 100.0 << endl;
    cout.unsetf(ios::showpoint);

    // Output a '+' before positive values
    cout.setf(ios::showpos);
    cout << 100 << " " << -100 << endl;
    cout.unsetf(ios::showpos);

    // Output numerical values in hexadecimal
    cout << hex << 100 << " " << 1000 << " " << 10000 << dec << endl;
}
```

6.4 Knuth-Morris-Pratt

```
/*
Finds all occurrences of the pattern string p within the
text string t. Running time is O(n + m), where n and m
are the lengths of p and t, respectively.
*/

#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef vector<int> VI;

void buildPi(string& p, VI& pi)
{
    pi = VI(p.length());
    int k = -2;
    for(int i = 0; i < p.length(); i++) {
        while(k >= -1 && p[k+1] != p[i])
            k = (k == -1) ? -2 : pi[k];
    }
}
```

```

    pi[i] = ++k;
}
}

int KMP(string& t, string& p)
{
    VI pi;
    buildPi(p, pi);
    int k = -1;
    for(int i = 0; i < t.length(); i++) {
        while(k >= -1 && p[k+1] != t[i]) {
            k = (k == -1) ? -2 : pi[k];
        }
        k++;
        if(k == p.length() - 1) {
            // p matches t[i-m+1, ..., i]
            cout << "matched at index " << i-k << ": ";
            cout << t.substr(i-k, p.length()) << endl;
            k = (k == -1) ? -2 : pi[k];
        }
    }
    return 0;
}

int main()
{
    string a = "AABAACAADAABAABA", b = "AABA";
    KMP(a, b); // expected matches at: 0, 9, 12
    return 0;
}

```

6.5 Topological sort (C++)

```

// This function uses performs a non-recursive topological sort.
//
// Running time:  $O(|V|^2)$ . If you use adjacency lists (vector<map<int> >),
// the running time is reduced to  $O(|E|)$ .
//
// INPUT:  w[i][j] = 1 if i should come before j, 0 otherwise
// OUTPUT: a permutation of 0,...,n-1 (stored in a vector)
//         which represents an ordering of the nodes which
//         is consistent with w
//
// If no ordering is possible, false is returned.

#include <iostream>
#include <queue>
#include <cmath>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;

bool TopologicalSort (const VVI &w, VI &order) {
    int n = w.size();
    VI parents (n);
    queue<int> q;
    order.clear();

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (w[j][i]) parents[i]++;
            if (parents[i] == 0) q.push (i);
        }
    }

    while (q.size() > 0) {
        int i = q.front();
        q.pop();
        order.push_back (i);
        for (int j = 0; j < n; j++) if (w[i][j]) {
            parents[j]--;
            if (parents[j] == 0) q.push (j);
        }
    }

    return (order.size() == n);
}

```

6.6 Random STL stuff

```

// Example for using stringstream and next_permutation

#include <algorithm>
#include <iostream>
#include <sstream>
#include <vector>

using namespace std;

int main(void) {
    vector<int> v;

    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);

    // Expected output: 1 2 3 4
    //                  1 2 4 3
    //                  ...
    //                  4 3 2 1
    do {
        stringstream oss;
        oss << v[0] << " " << v[1] << " " << v[2] << " " << v[3];

        // for input from a string s,
        // istream iss(s);
        // iss >> variable;

        cout << oss.str() << endl;
    } while (next_permutation (v.begin(), v.end()));

    v.clear();

    v.push_back(1);
    v.push_back(2);
    v.push_back(1);
    v.push_back(3);

    // To use unique, first sort numbers. Then call
    // unique to place all the unique elements at the beginning
    // of the vector, and then use erase to remove the duplicate
    // elements.

    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());

    // Expected output: 1 2 3
    for (size_t i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;
}

```

6.7 Fast exponentiation

```

/*
Uses powers of two to exponentiate numbers and matrices. Calculates
 $n^k$  in  $O(\log(k))$  time when  $n$  is a number. If  $A$  is an  $n \times n$  matrix,
calculates  $A^k$  in  $O(n^3 \log(k))$  time.
*/

#include <iostream>
#include <vector>

using namespace std;

typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T power(T x, int k) {
    T ret = 1;

    while(k) {
        if(k & 1) ret *= x;
        k >>= 1; x *= x;
    }
    return ret;
}

VVT multiply(VVT& A, VVT& B) {
    int n = A.size(), m = A[0].size(), k = B[0].size();
}

```

```

VVT C(n, VT(k, 0));

for(int i = 0; i < n; i++)
    for(int j = 0; j < k; j++)
        for(int l = 0; l < m; l++)
            C[i][j] += A[i][l] * B[l][j];

return C;
}

VVT power(VVT& A, int k) {
    int n = A.size();
    VVT ret(n, VT(n));
    for(int i = 0; i < n; i++) ret[i][i]=1;

    while(k) {
        if(k & 1) ret = multiply(ret, B);
        k >>= 1; B = multiply(B, B);
    }
    return ret;
}

int main()
{
    /* Expected Output:
    2.37^48 = 9.72569e+17

    376 264 285 220 265
    550 376 529 285 484
    484 265 376 264 285
    285 220 265 156 264
    529 285 484 265 376 */
    double n = 2.37;
    int k = 48;

    cout << n << "^" << k << " = " << power(n, k) << endl;

    double At[5][5] = {
        { 0, 0, 1, 0, 0 },
        { 1, 0, 0, 1, 0 },
        { 0, 0, 0, 0, 1 },
        { 1, 0, 0, 0, 0 },
        { 0, 1, 0, 0, 0 } };

    vector <vector <double> > A(5, vector <double>(5));
    for(int i = 0; i < 5; i++)
        for(int j = 0; j < 5; j++)
            A[i][j] = At[i][j];

    vector <vector <double> > Ap = power(A, k);

    cout << endl;
    for(int i = 0; i < 5; i++) {
        for(int j = 0; j < 5; j++)
            cout << Ap[i][j] << " ";
        cout << endl;
    }
}

```

6.8 Longest common subsequence

```

/*
Calculates the length of the longest common subsequence of two vectors.
Backtracks to find a single subsequence or all subsequences. Runs in
O(m*n) time except for finding all longest common subsequences, which
may be slow depending on how many there are.
*/

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;

typedef int T;
typedef vector<T> VT;
typedef vector<VT> VVT;

typedef vector<int> VI;
typedef vector<VI> VVI;

void backtrack(VVI& dp, VT& res, VT& A, VT& B, int i, int j)
{
    if(!i || !j) return;
    if(A[i-1] == B[j-1]) { res.push_back(A[i-1]); backtrack(dp, res, A, B, i-1, j-1); }
    else

```

```

{
    if(dp[i][j-1] >= dp[i-1][j]) backtrack(dp, res, A, B, i, j-1);
    else backtrack(dp, res, A, B, i-1, j);
}

void backtrackall(VVI& dp, set<VT>& res, VT& A, VT& B, int i, int j)
{
    if(!i || !j) { res.insert(VI()); return; }
    if(A[i-1] == B[j-1])
    {
        set<VT> tempres;
        backtrackall(dp, tempres, A, B, i-1, j-1);
        for(set<VT>::iterator it=tempres.begin(); it!=tempres.end(); it++)
        {
            VT temp = *it;
            temp.push_back(A[i-1]);
            res.insert(temp);
        }
    }
    else
    {
        if(dp[i][j-1] >= dp[i-1][j]) backtrackall(dp, res, A, B, i, j-1);
        if(dp[i][j-1] <= dp[i-1][j]) backtrackall(dp, res, A, B, i-1, j);
    }
}

VT LCS(VT& A, VT& B)
{
    VVI dp;
    int n = A.size(), m = B.size();
    dp.resize(n+1);
    for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);

    for(int i=1; i<=n; i++)
        for(int j=1; j<=m; j++)
        {
            if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
            else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }

    VT res;
    backtrack(dp, res, A, B, n, m);
    reverse(res.begin(), res.end());
    return res;
}

set<VT> LCSall(VT& A, VT& B)
{
    VVI dp;
    int n = A.size(), m = B.size();
    dp.resize(n+1);
    for(int i=0; i<=n; i++) dp[i].resize(m+1, 0);
    for(int i=1; i<=n; i++)
        for(int j=1; j<=m; j++)
        {
            if(A[i-1] == B[j-1]) dp[i][j] = dp[i-1][j-1]+1;
            else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }

    set<VT> res;
    backtrackall(dp, res, A, B, n, m);
    return res;
}

int main()
{
    int a[] = { 0, 5, 5, 2, 1, 4, 2, 3 }, b[] = { 5, 2, 4, 3, 2, 1, 2, 1, 3 };
    VI A = VI(a, a+8), B = VI(b, b+9);
    VI C = LCS(A, B);

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

    set<VI> D = LCSall(A, B);
    for(set<VI>::iterator it = D.begin(); it != D.end(); it++)
    {
        for(int i=0; i<(*it).size(); i++) cout << (*it)[i] << " ";
        cout << endl;
    }
}

```

6.9 Miller-Rabin Primality Test (C)

```

// Randomized Primality Test (Miller-Rabin):
// Error rate: 2^(-TRIAL)
// Almost constant time. srand is needed

```

```

#include <stdlib.h>
#define EPS 1e-7

typedef long long LL;

LL ModularMultiplication(LL a, LL b, LL m)
{
    LL ret=0, c=a;
    while(b)
    {
        if(b&1) ret=(ret+c)%m;
        b>>=1; c=(c+c)%m;
    }
    return ret;
}

LL ModularExponentiation(LL a, LL n, LL m)
{
    LL ret=1, c=a;
    while(n)
    {
        if(n&1) ret=ModularMultiplication(ret, c, m);
        n>>=1; c=ModularMultiplication(c, c, m);
    }
    return ret;
}

bool Witness(LL a, LL n)
{
    LL u=n-1;
    int t=0;
    while(!(u&1)) {u>>=1; t++;}

```

```

    LL x0=ModularExponentiation(a, u, n), x1;
    for(int i=1;i<=t;i++)
    {
        x1=ModularMultiplication(x0, x0, n);
        if(x1==1 && x0!=1 && x0!=n-1) return true;
        x0=x1;
    }
    if(x0!=1) return true;
    return false;
}

LL Random(LL n)
{
    LL ret=rand(); ret*=32768;
    ret+=rand(); ret*=32768;
    ret+=rand(); ret*=32768;
    ret+=rand();
    return ret%n;
}

bool IsPrimeFast(LL n, int TRIAL)
{
    while(TRIAL--)
    {
        LL a=Random(n-2)+1;
        if(Witness(a, n)) return false;
    }
    return true;
}

```
