

Technion Rubber Duck Forces Team Notebook

Contents

1 Combinatorial optimization	1
1.1 Sparse max-flow	1
1.2 Min-cost max-flow	2
1.3 Push-relabel max-flow	2
1.4 Min-cost matching	3
1.5 Max bipartite machine	4
1.6 Global min-cut	4
1.7 Graph cut inference	5
2 Geometry	6
2.1 Convex hull	6
2.2 Miscellaneous geometry	6
2.3 3D geometry	8
2.4 Slow Delaunay triangulation	8
3 Numerical algorithms	9
3.1 Eratosthenes Sieve	9
3.2 Number theory (modular, Chinese remainder, linear Diophantine)	9
3.3 Systems of linear equations, matrix inverse, determinant	10
3.4 Reduced row echelon form, matrix rank	10
3.5 Fast Fourier transform	11
3.6 Simplex algorithm	12
4 Graph algorithms	12
4.1 Fast Dijkstra's algorithm	12
4.2 Strongly connected components	13
4.3 Eulerian path	13
5 Data structures	13
5.1 Aho Corasick	13
5.2 Suffix array	14
5.3 Binary Indexed Tree	15
5.4 Union-find set	15
5.5 KD-tree	15
5.6 Splay tree	16
5.7 Fast segment tree	17
5.8 Fenwick tree	17
5.9 Lazy segment tree	18
5.10 Lowest common ancestor	18
5.11 Treap	19
5.12 Ukkonen	19
5.13 Z-Function	19
6 Miscellaneous	19
6.1 Longest increasing subsequence	19
6.2 Dates	20
6.3 Regular expressions	20
6.4 Prime numbers	21
6.5 C++ input/output	21
6.6 Knuth-Morris-Pratt	21
6.7 Latitude/longitude	21
6.8 Fast exponentiation	22
6.9 SAT-2	22
6.10 Ternary Search	23

1 Combinatorial optimization

1.1 Sparse max-flow

```

// Adjacency list implementation of Dinic's blocking flow algorithm.
// This is very fast in practice, and only loses to push-relabel flow.
//
// Running time:
//  $O(|V|^2 |E|)$ 
//
// INPUT:
// - graph, constructed using AddEdge()
// - source and sink
//
// OUTPUT:
// - maximum flow value
// - To obtain actual flow values, look at edges with capacity > 0
//   (zero capacity edges are residual edges).

#include<cstdio>
#include<vector>
#include<queue>
using namespace std;
typedef long long LL;

struct Edge {
    int u, v;
    LL cap, flow;
    Edge() {}
    Edge(int u, int v, LL cap): u(u), v(v), cap(cap), flow(0) {}
};

struct Dinic {
    int N;
    vector<Edge> E;
    vector<vector<int>>> g;
    vector<int> d, pt;

    Dinic(int N): N(N), E(0), g(N), d(N), pt(N) {}

    void AddEdge(int u, int v, LL cap) {
        if (u != v) {
            E.emplace_back(Edge(u, v, cap));
            g[u].emplace_back(E.size() - 1);
            E.emplace_back(Edge(v, u, 0));
            g[v].emplace_back(E.size() - 1);
        }
    }

    bool BFS(int S, int T) {
        queue<int> q({S});
        fill(d.begin(), d.end(), N + 1);
        d[S] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            if (u == T) break;
            for (int k: g[u]) {
                Edge &e = E[k];
                if (e.flow < e.cap && d[e.v] > d[u] + 1) {
                    d[e.v] = d[u] + 1;
                    q.emplace(e.v);
                }
            }
        }
        return d[T] != N + 1;
    }

    LL DFS(int u, int T, LL flow = -1) {
        if (u == T || flow == 0) return flow;
        for (int &i = pt[u]; i < g[u].size(); ++i) {
            Edge &e = E[g[u][i]];
            Edge &oe = E[g[u][i] + 1];
            if (d[e.v] == d[u] + 1) {
                LL amt = e.cap - e.flow;
                if (flow != -1 && amt > flow) amt = flow;
                if (LL pushed = DFS(e.v, T, amt)) {
                    e.flow += pushed;
                    oe.flow -= pushed;
                    return pushed;
                }
            }
        }
        return 0;
    }

    LL MaxFlow(int S, int T) {
        LL total = 0;
        while (BFS(S, T)) {
            fill(pt.begin(), pt.end(), 0);
            while (LL flow = DFS(S, T))
                total += flow;
        }
        return total;
    }
};

```

```
// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main()
{
    int N, E;
    scanf("%d%d", &N, &E);
    Dinic dinic(N);
    for(int i = 0; i < E; i++)
    {
        int u, v;
        LL cap;
        scanf("%d%d%lld", &u, &v, &cap);
        dinic.AddEdge(u - 1, v - 1, cap);
        dinic.AddEdge(v - 1, u - 1, cap);
    }
    printf("%lld\n", dinic.MaxFlow(0, N - 1));
    return 0;
}

// END CUT
```

1.2 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:  $O(|V|^4 * MAX\_EDGE\_COST)$  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> VPII;

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

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

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

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

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

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

```
width[s] = INF;

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

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

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

// 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
```

1.3 Push-relabel max-flow

```
// Adjacency list implementation of FIFO push relabel maximum flow
// with the gap relabeling heuristic. This implementation is
// significantly faster than straight Ford-Fulkerson. It solves
// random problems with 10000 vertices and 1000000 edges in a few
// seconds, though it is possible to construct test cases that
// achieve the worst-case.

// Running time:
//  $O(|V|^3)$ 
//
// INPUT:
// - graph, constructed using AddEdge()
// - source
// - sink
//
```

```

// OUTPUT:
// - maximum flow value
// - To obtain the actual flow values, look at all edges with
// capacity > 0 (zero capacity edges are residual edges).

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

using namespace std;

typedef long long LL;

struct Edge {
    int from, to, cap, flow, index;
    Edge(int from, int to, int cap, int flow, int index) :
        from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct PushRelabel {
    int N;
    vector<vector<Edge>> > G;
    vector<LL> excess;
    vector<int> dist, active, count;
    queue<int> Q;

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

    void AddEdge(int from, int to, int cap) {
        G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
        if (from == to) G[from].back().index++;
        G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
    }

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

    void Push(Edge &e) {
        int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
        if (dist[e.from] <= dist[e.to] || amt == 0) return;
        e.flow += amt;
        G[e.to][e.index].flow -= amt;
        excess[e.to] += amt;
        excess[e.from] -= amt;
        Enqueue(e.to);
    }

    void Gap(int k) {
        for (int v = 0; v < N; v++) {
            if (dist[v] < k) continue;
            count[dist[v]]--;
            dist[v] = max(dist[v], N+1);
            count[dist[v]]++;
            Enqueue(v);
        }
    }

    void Relabel(int v) {
        count[dist[v]]--;
        dist[v] = 2*N;
        for (int i = 0; i < G[v].size(); i++)
            if (G[v][i].cap - G[v][i].flow > 0)
                dist[v] = min(dist[v], dist[G[v][i].to] + 1);
        count[dist[v]]++;
        Enqueue(v);
    }

    void Discharge(int v) {
        for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i]);
        if (excess[v] > 0) {
            if (count[dist[v]] == 1)
                Gap(dist[v]);
            else
                Relabel(v);
        }
    }

    LL GetMaxFlow(int s, int t) {
        count[0] = N-1;
        count[N] = 1;
        dist[s] = N;
        active[s] = active[t] = true;
        for (int i = 0; i < G[s].size(); i++) {
            excess[s] += G[s][i].cap;
            Push(G[s][i]);
        }

        while (!Q.empty()) {
            int v = Q.front();
            Q.pop();

```

```

        active[v] = false;
        Discharge(v);
    }

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

// BEGIN CUT
// The following code solves SPOJ problem #4110: Fast Maximum Flow (FASTFLOW)

int main() {
    int n, m;
    scanf("%d%d", &n, &m);

    PushRelabel pr(n);
    for (int i = 0; i < m; i++) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        if (a == b) continue;
        pr.AddEdge(a-1, b-1, c);
        pr.AddEdge(b-1, a-1, c);
    }
    printf("%d\n", pr.GetMaxFlow(0, n-1));
    return 0;
}

// END CUT

```

1.4 Min-cost matching

```

////////////////////////////////////
// Min cost bipartite matching via shortest augmenting paths
//
// This is an O(n^3) implementation of a shortest augmenting path
// algorithm for finding min cost perfect matchings in dense
// graphs. In practice, it solves 1000x1000 problems in around 1
// second.
//
// cost[i][j] = cost for pairing left node i with right node j
// Lmate[i] = index of right node that left node i pairs with
// Rmate[j] = index of left node that right node j pairs with
//
// The values in cost[i][j] may be positive or negative. To perform
// maximization, simply negate the cost[i][j] matrix.
////////////////////////////////////

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

typedef vector<double> VVD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI &Lmate, VI &Rmate) {
    int n = int(cost.size());

    // construct dual feasible solution
    VVD u(n);
    VVD v(n);
    for (int i = 0; i < n; i++) {
        u[i] = cost[i][0];
        for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
    }
    for (int j = 0; j < n; j++) {
        v[j] = cost[0][j] - u[0];
        for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
    }

    // construct primal solution satisfying complementary slackness
    Lmate = VI(n, -1);
    Rmate = VI(n, -1);
    int mated = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (Rmate[j] != -1) continue;
            if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
                Lmate[i] = j;
                Rmate[j] = i;
                mated++;
                break;
            }

```

```

    }
}
}
VD dist(n);
VI dad(n);
VI seen(n);

// repeat until primal solution is feasible
while (mated < n) {

    // find an unmatched left node
    int s = 0;
    while (Lmate[s] != -1) s++;

    // initialize Dijkstra
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for (int k = 0; k < n; k++)
        dist[k] = cost[s][k] - u[s] - v[k];

    int j = 0;
    while (true) {

        // find closest
        j = -1;
        for (int k = 0; k < n; k++) {
            if (seen[k]) continue;
            if (j == -1 || dist[k] < dist[j]) j = k;
        }
        seen[j] = 1;

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

        // relax neighbors
        const int i = Rmate[j];
        for (int k = 0; k < n; k++) {
            if (seen[k]) continue;
            const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
            if (dist[k] > new_dist) {
                dist[k] = new_dist;
                dad[k] = j;
            }
        }

        // update dual variables
        for (int k = 0; k < n; k++) {
            if (k == j || !seen[k]) continue;
            const int i = Rmate[k];
            v[k] += dist[k] - dist[j];
            u[i] -= dist[k] - dist[j];
        }
        u[s] += dist[j];

        // augment along path
        while (dad[j] >= 0) {
            const int d = dad[j];
            Rmate[j] = Rmate[d];
            Lmate[Rmate[j]] = j;
            j = d;
        }
        Rmate[j] = s;
        Lmate[s] = j;

        mated++;
    }

    double value = 0;
    for (int i = 0; i < n; i++)
        value += cost[i][Lmate[i]];

    return value;
}

```

1.5 Max bipartite machine

```

// This code performs maximum bipartite matching.
//
// Running time:  $O(|E| |V|)$  -- often much faster in practice
//
// INPUT: w[i][j] = edge between row node i and column node j
// OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//         mc[j] = assignment for column node j, -1 if unassigned
//         function returns number of matches made

```

```

#include <vector>

using namespace std;

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

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
    for (int j = 0; j < w[i].size(); j++) {
        if (w[i][j] && !seen[j]) {
            seen[j] = true;
            if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
                mr[i] = j;
                mc[j] = i;
                return true;
            }
        }
    }
    return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
    mr = VI(w.size(), -1);
    mc = VI(w[0].size(), -1);

    int ct = 0;
    for (int i = 0; i < w.size(); i++) {
        VI seen(w[0].size());
        if (FindMatch(i, w, mr, mc, seen)) ct++;
    }
    return ct;
}

```

1.6 Global min-cut

```

// Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
//
// Running time:
//  $O(|V|^3)$ 
//
// INPUT:
// - graph, constructed using AddEdge()
//
// OUTPUT:
// - (min cut value, nodes in half of min cut)

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

using namespace std;

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

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights) {
    int N = weights.size();
    VI used(N), cut, best_cut;
    int best_weight = -1;

    for (int phase = N-1; phase >= 0; phase--) {
        VI w = weights[0];
        VI added = used;
        int prev, last = 0;
        for (int i = 0; i < phase; i++) {
            prev = last;
            last = -1;
            for (int j = 1; j < N; j++)
                if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
            if (i == phase-1) {
                for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
                for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
                used[last] = true;
                cut.push_back(last);
                if (best_weight == -1 || w[last] < best_weight) {
                    best_weight = cut;
                    best_weight = w[last];
                }
            }
            else {
                for (int j = 0; j < N; j++)
                    w[j] += weights[last][j];
                added[last] = true;
            }
        }
    }
    return make_pair(best_weight, best_cut);
}

```

```

}
// BEGIN CUT
// The following code solves UVA problem #10989: Bomb, Divide and Conquer
int main() {
    int N;
    cin >> N;
    for (int i = 0; i < N; i++) {
        int n, m;
        cin >> n >> m;
        VVI weights(n, VI(n));
        for (int j = 0; j < m; j++) {
            int a, b, c;
            cin >> a >> b >> c;
            weights[a-1][b-1] = weights[b-1][a-1] = c;
        }
        pair<int, VI> res = GetMinCut(weights);
        cout << "Case #" << i+1 << ": " << res.first << endl;
    }
}
// END CUT

```

1.7 Graph cut inference

```

// Special-purpose {0,1} combinatorial optimization solver for
// problems of the following by a reduction to graph cuts:
//
//      minimize          sum_i psi_i(x[i])
//      x[1]...x[n] in {0,1}      + sum_{i < j} phi_{ij}(x[i], x[j])
//
// where
//      psi_i : {0, 1} --> R
//      phi_{ij} : {0, 1} x {0, 1} --> R
//
// such that
//      phi_{ij}(0,0) + phi_{ij}(1,1) <= phi_{ij}(0,1) + phi_{ij}(1,0)  (*)
//
// This can also be used to solve maximization problems where the
// direction of the inequality in (*) is reversed.
//
// INPUT: phi -- a matrix such that phi[i][j][u][v] = phi_{ij}(u, v)
//        psi -- a matrix such that psi[i][u] = psi_i(u)
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution
//
// To use this code, create a GraphCutInference object, and call the
// DoInference() method. To perform maximization instead of minimization,
// ensure that #define MAXIMIZATION is enabled.

#include <vector>
#include <iostream>

using namespace std;

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

const int INF = 1000000000;

// comment out following line for minimization
#define MAXIMIZATION

struct GraphCutInference {
    int N;
    VVI cap, flow;
    VI reached;

    int Augment(int s, int t, int a) {
        reached[s] = 1;
        if (s == t) return a;
        for (int k = 0; k < N; k++) {
            if (reached[k]) continue;
            if (int aa = min(a, cap[s][k] - flow[s][k])) {
                if (int b = Augment(k, t, aa)) {
                    flow[s][k] += b;
                    flow[k][s] -= b;
                    return b;
                }
            }
        }
        return 0;
    }

    int GetMaxFlow(int s, int t) {

```

```

        N = cap.size();
        flow = VVI(N, VI(N));
        reached = VI(N);

        int totflow = 0;
        while (int amt = Augment(s, t, INF)) {
            totflow += amt;
            fill(reached.begin(), reached.end(), 0);
        }
        return totflow;
    }

    int DoInference(const VVVVI &phi, const VVI &psi, VI &x) {
        int M = phi.size();
        cap = VVI(M+2, VI(M+2));
        VI b(M);
        int c = 0;

        for (int i = 0; i < M; i++) {
            b[i] += psi[i][1] - psi[i][0];
            c += psi[i][0];
            for (int j = 0; j < i; j++)
                b[i] += phi[i][j][1][1] - phi[i][j][0][1];
            for (int j = i+1; j < M; j++) {
                cap[i][j] = phi[i][j][0][1] + phi[i][j][1][0] - phi[i][j][0][0] - phi[i][j][1][1];
                b[i] += phi[i][j][1][0] - phi[i][j][0][0];
                c += phi[i][j][0][0];
            }
        }

#ifdef MAXIMIZATION
        for (int i = 0; i < M; i++) {
            for (int j = i+1; j < M; j++)
                cap[i][j] *= -1;
            b[i] *= -1;
        }
        c *= -1;
#endif

        for (int i = 0; i < M; i++) {
            if (b[i] >= 0) {
                cap[M][i] = b[i];
            } else {
                cap[i][M+1] = -b[i];
                c += b[i];
            }
        }

        int score = GetMaxFlow(M, M+1);
        fill(reached.begin(), reached.end(), 0);
        Augment(M, M+1, INF);
        x = VI(M);
        for (int i = 0; i < M; i++) x[i] = reached[i] ? 0 : 1;
        score += c;
#ifdef MAXIMIZATION
        score *= -1;
#endif

        return score;
    }

};

int main() {
    // solver for "Cat vs. Dog" from NWERC 2008

    int numcases;
    cin >> numcases;
    for (int caseno = 0; caseno < numcases; caseno++) {
        int c, d, v;
        cin >> c >> d >> v;

        VVVVI phi(c+d, VVVI(c+d, VVI(2, VI(2))));
        VVI psi(c+d, VI(2));
        for (int i = 0; i < v; i++) {
            char p, q;
            int u, v;
            cin >> p >> u >> q >> v;
            u--; v--;
            if (p == 'C') {
                phi[u][c+v][0][0]++;
                phi[c+v][u][0][0]++;
            } else {
                phi[v][c+u][1][1]++;
                phi[c+u][v][1][1]++;
            }
        }

        GraphCutInference graph;
        VI x;
        cout << graph.DoInference(phi, psi, x) << endl;
    }
}

```

```

    }
    return 0;
}

```

2 Geometry

2.1 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 <cstdio>
#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

```

2.2 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) {
    os << "(" << p.x << ", " << p.y << ")";
}

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

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

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

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

```

```

double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

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

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

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

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

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

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
    b=(a+b)/2;
    c=(a+c)/2;
    return ComputeLineIntersection(b, b+RotateCCW90(a-b), c, c+RotateCCW90(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].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
    for (int i = 0; i < p.size(); i++) {
        for (int k = i+1; k < p.size(); k++) {
            int j = (i+1) % p.size();
            int l = (k+1) % p.size();
            if (i == l || j == k) continue;
            if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

inline bool cw(const PT &from, const PT &to) { return cross(from, to) < -EPS; }
inline bool ccw(const PT &from, const PT &to) { return cross(from, to) > EPS; }

// CW
inline bool isInsideTriangle(const PT &point, const PT triangle[])
{
    const int n = 3;
    for (int i = 0; i < n; ++i)
    {
        if (cw(point - triangle[i], triangle[(i+1) % n] - triangle[i]))
            return false;
    }
    return true;
}

// CW
inline bool isInsideHull(const PT &point, const int hullSize, const PT hull[])
{
    int bottomNeighbourIndex = (int)(lower_bound(hull + 2, hull + hullSize, point, [&](const PT &
        current, const PT &needle) {
            return ccw(needle - hull[0], current - hull[0]);
        }) - hull);
    if (bottomNeighbourIndex >= hullSize)
    {
        return false;
    }
}

```

```

    }

    const PT triangle[] = { hull[0], hull[bottomNeighbourIndex-1], hull[bottomNeighbourIndex] };

    return isInsideTriangle(point, triangle);
}

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(4.5,4.5), 10, sqrt(2.0)/2.0);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

    // area should be 5.0
    // centroid should be (1.1666666, 1.166666)

```

```

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

return 0;
}

```

2.3 3D geometry

```

public class Geom3D {
    // distance from point (x, y, z) to plane aX + bY + cZ + d = 0
    public static double ptPlaneDist(double x, double y, double z,
        double a, double b, double c, double d) {
        return Math.abs(a*x + b*y + c*z + d) / Math.sqrt(a*a + b*b + c*c);
    }

    // distance between parallel planes aX + bY + cZ + d1 = 0 and
    // aX + bY + cZ + d2 = 0
    public static double planePlaneDist(double a, double b, double c,
        double d1, double d2) {
        return Math.abs(d1 - d2) / Math.sqrt(a*a + b*b + c*c);
    }

    // distance from point (px, py, pz) to line (x1, y1, z1)-(x2, y2, z2)
    // (or ray, or segment; in the case of the ray, the endpoint is the
    // first point)
    public static final int LINE = 0;
    public static final int SEGMENT = 1;
    public static final int RAY = 2;
    public static double ptLineDistSq(double x1, double y1, double z1,
        double x2, double y2, double z2, double px, double py, double pz,
        int type) {
        double pd2 = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2) + (z1-z2)*(z1-z2);

        double x, y, z;
        if (pd2 == 0) {
            x = x1;
            y = y1;
            z = z1;
        } else {
            double u = ((px-x1)*(x2-x1) + (py-y1)*(y2-y1) + (pz-z1)*(z2-z1)) / pd2;
            x = x1 + u * (x2 - x1);
            y = y1 + u * (y2 - y1);
            z = z1 + u * (z2 - z1);
            if (type != LINE && u < 0) {
                x = x1;
                y = y1;
                z = z1;
            }
            if (type == SEGMENT && u > 1.0) {
                x = x2;
                y = y2;
                z = z2;
            }
        }

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

    public static double ptLineDist(double x1, double y1, double z1,
        double x2, double y2, double z2, double px, double py, double pz,
        int type) {
        return Math.sqrt(ptLineDistSq(x1, y1, z1, x2, y2, z2, px, py, pz, type));
    }
}

```

2.4 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;
}

```

3 Numerical algorithms

3.1 Eratosthenes Sieve

```

int n;
vector<char> prime (n+1, true);
prime[0] = prime[1] = false;
for (int i=2; i<=n; ++i)
    if (prime[i])
        if (i * 1ll * i <= n)
            for (int j=i*i; j<=n; j+=i)
                prime[j] = false;

```

3.2 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;
}

```

3.3 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 (pj == -1 || 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;
        }
    }
}

```

3.4 Reduced row echelon form, matrix rank

```

// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting. This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//

```

```

// INPUT:  a[] = an nxm matrix
//
// OUTPUT: rref[][] = an nxm matrix (stored in a[][])
// returns rank of a[][]

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

using namespace std;

const double EPSILON = 1e-10;

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

int rref(VVT &a) {
    int n = a.size();
    int m = a[0].size();
    int r = 0;
    for (int c = 0; c < m && r < n; c++) {
        int j = r;
        for (int i = r + 1; i < n; i++)
            if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
        if (fabs(a[j][c]) < EPSILON) continue;
        swap(a[j], a[r]);

        T s = 1.0 / a[r][c];
        for (int j = 0; j < m; j++) a[r][j] *= s;
        for (int i = 0; i < n; i++) if (i != r) {
            T t = a[i][c];
            for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
        }
        r++;
    }
    return r;
}

int main() {
    const int n = 5, m = 4;
    double A[n][m] = {
        {16, 2, 3, 13},
        {5, 11, 10, 8},
        {9, 7, 6, 12},
        {4, 14, 15, 1},
        {13, 21, 21, 13}};
    VVT a(n);
    for (int i = 0; i < n; i++)
        a[i] = VT(A[i], A[i] + m);

    int rank = rref(a);

    // expected: 3
    cout << "Rank: " << rank << endl;

    // expected: 1 0 0 1
    // 0 1 0 3
    // 0 0 1 -3
    // 0 0 0 3.10862e-15
    // 0 0 0 2.22045e-15
    cout << "rref: " << endl;
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 4; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }
}

```

3.5 Fast Fourier transform

```

#include <cassert>
#include <cstdio>
#include <cmath>

struct cpx
{
    cpx() {}
    cpx(double aa):a(aa),b(0){}
    cpx(double aa, double bb):a(aa),b(bb){}
    double a;
    double b;
    double modsq(void) const
    {
        return a * a + b * b;
    }
    cpx bar(void) const
    {

```

```

        return cpx(a, -b);
    }
};

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

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

cpx operator /(cpx a, cpx b)
{
    cpx r = a * b.bar();
    return cpx(r.a / b.modsq(), r.b / b.modsq());
}

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

const double two_pi = 4 * acos(0);

// in:    input array
// out:   output array
// step:  {SET TO 1} (used internally)
// size:  length of the input/output {MUST BE A POWER OF 2}
// dir:   either plus or minus one (direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size-1} in[j] * exp(dir * 2pi * i * j * k / size)
void FFT(cpx *in, cpx *out, int step, int size, int dir)
{
    if (size < 1) return;
    if (size == 1)
    {
        out[0] = in[0];
        return;
    }
    FFT(in, out, step * 2, size / 2, dir);
    FFT(in + step, out + size / 2, step * 2, size / 2, dir);
    for (int i = 0; i < size / 2; i++)
    {
        cpx even = out[i];
        cpx odd = out[i + size / 2];
        out[i] = even + EXP(dir * two_pi * i / size) * odd;
        out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size) * odd;
    }
}

// Usage:
// f[0...N-1] and g[0...N-1] are numbers
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise product).
// To compute h[] in O(N log N) time, do the following:
// 1. Compute F and G (pass dir = 1 as the argument).
// 2. Get H by element-wise multiplying F and G.
// 3. Get h by taking the inverse FFT (use dir = -1 as the argument)
// and *dividing by N. DO NOT FORGET THIS SCALING FACTOR.

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

    cpx a[8] = {0, 1, cpx(1,3), cpx(0,5), 1, 0, 2, 0};
    cpx b[8] = {1, cpx(0,-2), cpx(0,1), 3, -1, -3, 1, -2};
    cpx A[8];
    cpx B[8];
    FFT(a, A, 1, 8, 1);
    FFT(b, B, 1, 8, 1);

    for (int i = 0; i < 8; i++)
    {
        printf("%7.21f%7.21f", A[i].a, A[i].b);
    }
    printf("\n");
    for (int i = 0; i < 8; i++)
    {
        cpx Ai(0,0);
        for (int j = 0; j < 8; j++)
        {
            Ai = Ai + a[j] * EXP(j * i * two_pi / 8);
        }
        printf("%7.21f%7.21f", Ai.a, Ai.b);
    }
    printf("\n");
}

```

```

cpx AB[8];
for(int i = 0 ; i < 8 ; i++)
    AB[i] = A[i] * B[i];
cpx aconvb[8];
FFT(AB, aconvb, 1, 8, -1);
for(int i = 0 ; i < 8 ; i++)
    aconvb[i] = aconvb[i] / 8;
for(int i = 0 ; i < 8 ; i++)
{
    printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);
}
printf("\n");
for(int i = 0 ; i < 8 ; i++)
{
    cpx aconvbi(0,0);
    for(int j = 0 ; j < 8 ; j++)
    {
        aconvbi = aconvbi + a[j] * b[(8 + i - j) % 8];
    }
    printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);
}
printf("\n");
return 0;
}

```

3.6 Simplex algorithm

```

// Two-phase simplex algorithm for solving linear programs of the form
//
//      maximize    c^T x
//      subject to  Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
//
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b, and c as
// arguments. Then, call Solve(x).

#include <iostream>
#include <iomanip>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
    int m, n;
    VI B, N;
    VVD D;

    LPSolver(const VVD &A, const VD &b, const VD &c) :
        m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, VD(n+2)) {
        for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
        for (int i = 0; i < m; i++) { B[i] = n+1; D[i][n] = -1; D[i][n+1] = b[i]; }
        for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m+1][n] = 1;
    }

    void Pivot(int r, int s) {
        double inv = 1.0 / D[r][s];
        for (int i = 0; i < m+2; i++) if (i != r)
            for (int j = 0; j < n+2; j++) if (j != s)
                D[i][j] -= D[r][j] * D[i][s] * inv;
        for (int j = 0; j < n+2; j++) if (j != s) D[r][j] *= inv;
        for (int i = 0; i < m+2; i++) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
    }

    bool Simplex(int phase) {
        int x = phase == 1 ? m+1 : m;
        while (true) {

```

```

            int s = -1;
            for (int j = 0; j <= n; j++) {
                if (phase == 2 && N[j] == -1) continue;
                if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
            }
            if (D[x][s] > -EPS) return true;
            int r = -1;
            for (int i = 0; i < m; i++) {
                if (D[i][s] < EPS) continue;
                if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
                    (D[i][n+1] / D[i][s]) == (D[r][n+1] / D[r][s]) && B[i] < B[r]) r = i;
            }
            if (r == -1) return false;
            Pivot(r, s);
        }
    }

    DOUBLE Solve(VD &x) {
        int r = 0;
        for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
        if (D[r][n+1] < -EPS) {
            Pivot(r, n);
            if (!Simplex(1)) return -numeric_limits<DOUBLE>::infinity();
            for (int i = 0; i < m; i++) if (B[i] == -1) {
                int s = -1;
                for (int j = 0; j <= n; j++)
                    if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
                Pivot(i, s);
            }
        }
        if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
        x = VD(n);
        for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
        return D[m][n+1];
    }
};

int main() {
    const int m = 4;
    const int n = 3;
    DOUBLE _A[m][n] = {
        { 6, -1, 0 },
        { -1, -5, 0 },
        { 1, 5, 1 },
        { -1, -5, -1 }
    };
    DOUBLE _b[m] = { 10, -4, 5, -5 };
    DOUBLE _c[n] = { 1, -1, 0 };

    VVD A(m);
    VD b(_b, _b + m);
    VD c(_c, _c + n);
    for (int i = 0; i < m; i++) A[i] = VD(_A[i], _A[i] + n);

    LPSolver solver(A, b, c);
    VD x;
    DOUBLE value = solver.Solve(x);

    cerr << "VALUE: " << value << endl; // VALUE: 1.29032
    cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
    for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
    cerr << endl;
    return 0;
}

```

4 Graph algorithms

4.1 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 1 2 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
*/

```

4.2 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]);}
}

```

4.3 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;
}

```

5 Data structures

5.1 Aho Corasick

```

template < int ALPHA >
class AhoCorasick
{
public:
    static const int ILLEGAL_INDEX;
    static const int ROOT;

    struct Node
    {
        bool leaf;
        int parent;
        int parentCharacter;
        int link;

        int next[ALPHA];
        int go[ALPHA];
        int outputFunction;

        Node(int parent = ILLEGAL_INDEX, int parentCharacter = ALPHA) :
            leaf(false),
            parent(parent),
            parentCharacter(parentCharacter),
            link(ILLEGAL_INDEX),
            outputFunction(ILLEGAL_INDEX)
        {
            fill_n(next, ALPHA, ILLEGAL_INDEX);
            fill_n(go, ALPHA, ILLEGAL_INDEX);

```

```

    }
};

vector<Node> tree = vector<Node>(1);

AhoCorasick() {}
AhoCorasick(int maxStatesNumber)
{
    tree.reserve(maxStatesNumber);
}

template < class Iterator >
void add(int length, const Iterator begin)
{
    int vertex = ROOT;
    for (int i = 0; i < length; ++i)
    {
        if (ILLEGAL_INDEX == tree[vertex].next[begin[i]])
        {
            tree[vertex].next[begin[i]] = SZ(tree);
            tree.push_back(Node(vertex, begin[i]));
        }

        vertex = tree[vertex].next[begin[i]];
    }

    tree[vertex].leaf = true;
}

int getLink(int vertex)
{
    assert(0 <= vertex && vertex < tree.size());

    if (ILLEGAL_INDEX == tree[vertex].link)
    {
        if (ROOT == vertex || ROOT == tree[vertex].parent)
        {
            tree[vertex].link = ROOT;
        }
        else
        {
            tree[vertex].link = go(getLink(tree[vertex].parent), tree[vertex].parentCharacter);
        }
    }

    return tree[vertex].link;
}

int go(int vertex, int character)
{
    assert(0 <= character && character < ALPHA);
    assert(0 <= vertex && vertex < tree.size());

    if (ILLEGAL_INDEX == tree[vertex].go[character])
    {
        if (ILLEGAL_INDEX == tree[vertex].next[character])
        {
            tree[vertex].go[character] = ROOT == vertex ? ROOT : go(getLink(vertex), character);
        }
        else
        {
            tree[vertex].go[character] = tree[vertex].next[character];
        }
    }

    return tree[vertex].go[character];
}

int getOutputFunction(int vertex)
{
    assert(0 <= vertex && vertex < tree.size());

    if (ILLEGAL_INDEX == tree[vertex].outputFunction)
    {
        if (tree[vertex].leaf || ROOT == vertex)
        {
            tree[vertex].outputFunction = vertex;
        }
        else
        {
            tree[vertex].outputFunction = getOutputFunction(getLink(vertex));
        }
    }

    return tree[vertex].outputFunction;
}

};

template < int ALPHA > const int AhoCorasick<ALPHA>::ILLEGAL_INDEX = -1;
template < int ALPHA > const int AhoCorasick<ALPHA>::ROOT = 0;

```

5.2 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

```

5.3 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;
}

```

5.4 Union-find set

```

// BEGIN CUT
#include <iostream>
#include <vector>
using namespace std;

// END CUT
struct UnionFind {
    vector< int > parent;
    vector< int > rank;
    UnionFind(int n) : parent(n), rank(n) {
        for (int i = 0; i < n; ++i) {
            parent[i] = i;

```

```

            rank[i] = 0;
        }
    }

    int find_set (int v) {
        if (v == parent[v])
            return v;
        return parent[v] = find_set (parent[v]);
    }

    void union_sets (int a, int b) {
        a = find_set (a);
        b = find_set (b);
        if (a != b) {
            if (rank[a] < rank[b])
                swap (a, b);
            parent [b] = a;
            if (rank[a] == rank[b])
                ++rank[a];
        }
    }
};
// BEGIN CUT

int main()
{
    int n = 5;
    UnionFind C(n);
    C.union_sets(0, 2);
    C.union_sets(1, 0);
    C.union_sets(3, 4);
    for (int i = 0; i < n; i++) cout << i << " " << C.find_set(i) << endl;
    return 0;
}
// END CUT

```

5.5 KD-tree

```

// -----
// A straightforward, but probably sub-optimal KD-tree implementation
// that's probably good enough for most things (current it's a
// 2D-tree)
//
// - constructs from n points in O(n lg^2 n) time
// - handles nearest-neighbor query in O(lg n) if points are well
//   distributed
// - worst case for nearest-neighbor may be linear in pathological
//   case
//
// Sonny Chan, Stanford University, April 2009
// -----

#include <iostream>
#include <vector>
#include <limits>
#include <cstdlib>

using namespace std;

// number type for coordinates, and its maximum value
typedef long long ntype;
const ntype sentry = numeric_limits<ntype>::max();

// point structure for 2D-tree, can be extended to 3D
struct point {
    ntype x, y;
    point (ntype xx = 0, ntype yy = 0) : x(xx), y(yy) {}
};

bool operator==(const point &a, const point &b)
{
    return a.x == b.x && a.y == b.y;
}

// sorts points on x-coordinate
bool on_x(const point &a, const point &b)
{
    return a.x < b.x;
}

// sorts points on y-coordinate
bool on_y(const point &a, const point &b)
{
    return a.y < b.y;
}

// squared distance between points
ntype pdist2(const point &a, const point &b)
{

```

```

    ntype dx = a.x-b.x, dy = a.y-b.y;
    return dx*dx + dy*dy;
}

// bounding box for a set of points
struct bbox
{
    ntype x0, x1, y0, y1;

    bbox() : x0(sentry), x1(-sentry), y0(sentry), y1(-sentry) {}

    // computes bounding box from a bunch of points
    void compute(const vector<point> &v) {
        for (int i = 0; i < v.size(); ++i) {
            x0 = min(x0, v[i].x);    x1 = max(x1, v[i].x);
            y0 = min(y0, v[i].y);    y1 = max(y1, v[i].y);
        }
    }

    // squared distance between a point and this bbox, 0 if inside
    ntype distance(const point &p) {
        if (p.x < x0) {
            if (p.y < y0)    return pdist2(point(x0, y0), p);
            else if (p.y > y1) return pdist2(point(x0, y1), p);
            else            return pdist2(point(x0, p.y), p);
        }
        else if (p.x > x1) {
            if (p.y < y0)    return pdist2(point(x1, y0), p);
            else if (p.y > y1) return pdist2(point(x1, y1), p);
            else            return pdist2(point(x1, p.y), p);
        }
        else {
            if (p.y < y0)    return pdist2(point(p.x, y0), p);
            else if (p.y > y1) return pdist2(point(p.x, y1), p);
            else            return 0;
        }
    }
};

// stores a single node of the kd-tree, either internal or leaf
struct kndode
{
    bool leaf;        // true if this is a leaf node (has one point)
    point pt;         // the single point of this is a leaf
    bbox bound;       // bounding box for set of points in children

    kndode *first, *second; // two children of this kd-node

    kndode() : leaf(false), first(0), second(0) {}
    ~kndode() { if (first) delete first; if (second) delete second; }

    // intersect a point with this node (returns squared distance)
    ntype intersect(const point &p) {
        return bound.distance(p);
    }

    // recursively builds a kd-tree from a given cloud of points
    void construct(vector<point> &vp)
    {
        // compute bounding box for points at this node
        bound.compute(vp);

        // if we're down to one point, then we're a leaf node
        if (vp.size() == 1) {
            leaf = true;
            pt = vp[0];
        }
        else {
            // split on x if the bbox is wider than high (not best heuristic...)
            if (bound.x1-bound.x0 >= bound.y1-bound.y0)
                sort(vp.begin(), vp.end(), on_x);
            // otherwise split on y-coordinate
            else
                sort(vp.begin(), vp.end(), on_y);

            // divide by taking half the array for each child
            // (not best performance if many duplicates in the middle)
            int half = vp.size()/2;
            vector<point> vl(vp.begin(), vp.begin()+half);
            vector<point> vr(vp.begin()+half, vp.end());
            first = new kndode();    first->construct(vl);
            second = new kndode();   second->construct(vr);
        }
    }
};

// simple kd-tree class to hold the tree and handle queries
struct kdtree
{
    kndode *root;

    // constructs a kd-tree from a points (copied here, as it sorts them)

```

```

kdtree(const vector<point> &vp) {
    vector<point> v(vp.begin(), vp.end());
    root = new kndode();
    root->construct(v);
}

~kdtree() { delete root; }

// recursive search method returns squared distance to nearest point
ntype search(kndode *node, const point &p)
{
    if (node->leaf) {
        // commented special case tells a point not to find itself
        if (p == node->pt) return sentry;
        else
            return pdist2(p, node->pt);
    }

    ntype bfirst = node->first->intersect(p);
    ntype bsecond = node->second->intersect(p);

    // choose the side with the closest bounding box to search first
    // (note that the other side is also searched if needed)
    if (bfirst < bsecond) {
        ntype best = search(node->first, p);
        if (bsecond < best)
            best = min(best, search(node->second, p));
        return best;
    }
    else {
        ntype best = search(node->second, p);
        if (bfirst < best)
            best = min(best, search(node->first, p));
        return best;
    }
}

// squared distance to the nearest
ntype nearest(const point &p) {
    return search(root, p);
}

// -----
// some basic test code here

int main()
{
    // generate some random points for a kd-tree
    vector<point> vp;
    for (int i = 0; i < 100000; ++i) {
        vp.push_back(point(rand()%100000, rand()%100000));
    }
    kdtree tree(vp);

    // query some points
    for (int i = 0; i < 10; ++i) {
        point q(rand()%100000, rand()%100000);
        cout << "Closest squared distance to (" << q.x << ", " << q.y << ") "
              << " is " << tree.nearest(q) << endl;
    }

    return 0;
}

// -----

```

5.6 Splay tree

```

#include <cstdio>
#include <algorithm>
using namespace std;

const int N_MAX = 130010;
const int oo = 0x3f3f3f3f;
struct Node
{
    Node *ch[2], *pre;
    int val, size;
    bool isTurned;
} nodePool[N_MAX], *null, *root;

Node *allocNode(int val)
{
    static int freePos = 0;
    Node *x = &nodePool[freePos++];
    x->val = val, x->isTurned = false;
    x->ch[0] = x->ch[1] = x->pre = null;

```



```

    x->size = 1;
    return x;
}

inline void update(Node *x)
{
    x->size = x->ch[0]->size + x->ch[1]->size + 1;
}

inline void makeTurned(Node *x)
{
    if(x == null)
        return;
    swap(x->ch[0], x->ch[1]);
    x->isTurned ^= 1;
}

inline void pushDown(Node *x)
{
    if(x->isTurned)
    {
        makeTurned(x->ch[0]);
        makeTurned(x->ch[1]);
        x->isTurned ^= 1;
    }
}

inline void rotate(Node *x, int c)
{
    Node *y = x->pre;
    x->pre = y->pre;
    if(y->pre != null)
        y->pre->ch[y == y->pre->ch[1]] = x;
    y->ch[!c] = x->ch[c];
    if(x->ch[c] != null)
        x->ch[c]->pre = y;
    x->ch[c] = y, y->pre = x;
    update(y);
    if(y == root)
        root = x;
}

void splay(Node *x, Node *p)
{
    while(x->pre != p)
    {
        if(x->pre->pre == p)
            rotate(x, x == x->pre->ch[0]);
        else
        {
            Node *y = x->pre, *z = y->pre;
            if(y == z->ch[0])
            {
                if(x == y->ch[0])
                    rotate(y, 1), rotate(x, 1);
                else
                    rotate(x, 0), rotate(x, 1);
            }
            else
            {
                if(x == y->ch[1])
                    rotate(y, 0), rotate(x, 0);
                else
                    rotate(x, 1), rotate(x, 0);
            }
        }
        update(x);
    }
}

void select(int k, Node *fa)
{
    Node *now = root;
    while(1)
    {
        pushDown(now);
        int tmp = now->ch[0]->size + 1;
        if(tmp == k)
            break;
        else if(tmp < k)
            now = now->ch[1], k -= tmp;
        else
            now = now->ch[0];
    }
    splay(now, fa);
}

Node *makeTree(Node *p, int l, int r)
{
    if(l > r)
        return null;
    int mid = (l + r) / 2;

```

```

    Node *x = allocNode(mid);
    x->pre = p;
    x->ch[0] = makeTree(x, l, mid - 1);
    x->ch[1] = makeTree(x, mid + 1, r);
    update(x);
    return x;
}

int main()
{
    int n, m;
    null = allocNode(0);
    null->size = 0;
    root = allocNode(0);
    root->ch[1] = allocNode(oo);
    root->ch[1]->pre = root;
    update(root);

    scanf("%d%d", &n, &m);
    root->ch[1]->ch[0] = makeTree(root->ch[1], 1, n);
    splay(root->ch[1]->ch[0], null);

    while(m --)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        a ++, b ++;
        select(a - 1, null);
        select(b + 1, root);
        makeTurned(root->ch[1]->ch[0]);
    }

    for(int i = 1; i <= n; i ++)
    {
        select(i + 1, null);
        printf("%d ", root->val);
    }
}

```

5.7 Fast segment tree

```

//BEGIN CUT
#include <stdio>
//END CUT

struct SegmentTree{
    static const int N = 1e5; // limit for array size
    int n; // array size
    int t[2 * N];

    void build() { // build the tree
        for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
    }

    void modify(int p, int value) { // set value at position p
        for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
    }

    int query(int l, int r) { // sum on interval [l, r)
        int res = 0;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l&1) res += t[l++];
            if (r&1) res += t[--r];
        }
        return res;
    }
};

// BEGIN CUT

SegmentTree st;

int main() {
    scanf("%d", &st.n);
    for (int i = 0; i < st.n; ++i) scanf("%d", &st.t + st.n + i);
    st.build();
    st.modify(0, 1);
    printf("%d\n", st.query(3, 11));
    return 0;
}

// END CUT

```

5.8 Fenwick tree

```

vector<int> t;
int n;

void init (int nn) {
    n = nn;
    t.assign (n, 0);
}

int sum (int r) {
    int result = 0;
    for (; r >= 0; r = (r & (r+1)) - 1)
        result += t[r];
    return result;
}

void inc (int i, int delta) {
    for (; i < n; i = (i | (i+1)))
        t[i] += delta;
}

int sum (int l, int r) {
    return sum (r) - sum (l-1);
}

```

5.9 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;
    }
}

```

5.10 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])
            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;
}

```

5.11 Treap

```

struct item {
    int key, prior;
    int cnt;
    item * l, * r;
    item() {}
    item(int key, int prior) : key(key), prior(prior), cnt(0), l(NULL), r(NULL) {}
};
typedef item * pitem;

int cnt(pitem t) {
    return t ? t->cnt : 0;
}

void upd_cnt(pitem t) {
    if (t)
        t->cnt = 1 + cnt(t->l) + cnt(t->r);
}

void merge(pitem & t, pitem l, pitem 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 );
    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 insert(pitem & t, pitem it) {
    if (!t)
        t = it;
    else if (it->prior > t->prior)
        split(t, it->l, it->r, it->key), t = it;
    else
        insert(it->key < t->key ? t->l : t->r, it);
}

void erase(pitem & t, int key) {
    if (t->key == key)
        merge(t, t->l, t->r);
    else
        erase(key < t->key ? t->l : t->r, key);
}

```

5.12 Ukkonen

```

const int N=1000000, // maximum possible number of nodes in suffix tree
INF=1000000000000; // infinity constant
string a; // input string for which the suffix tree is being built
int t[N][26], // array of transitions (state, letter)
l[N], // left...
r[N], // ...and right boundaries of the substring of a which correspond to incoming edge
p[N], // parent of the node
s[N], // suffix link
tv, // the node of the current suffix (if we're mid-edge, the lower node of the edge)
tp, // position in the string which corresponds to the position on the edge (between l[tp] and r[tp], inclusive)
ts, // the number of nodes
ls; // the current character in the string

void ukkadd(int c) { // add character s to the tree
    suff++; // we'll return here after each transition to the suffix (and will add character again)
    if (r[tp]<tp) { // check whether we're still within the boundaries of the current edge

```

```

// if we're not, find the next edge. If it doesn't exist, create a leaf and add it to the tree
if (t[tp][c]==-1) {t[tp][c]=ts;l[ts]=ls;p[ts++]=tv;tv=s[tp];tp=r[tp]+1;goto suff;}
tv=t[tp][c];tp=l[tp];
} // otherwise just proceed to the next edge
if (tp==-1 || c==a[tp]-'a')
    tp++; // if the letter on the edge equal c, go down that edge
else {
    // otherwise split the edge in two with middle in node ts
    l[ts]=l[tp];r[ts]=tp-1;p[ts]=p[tp];t[ts][a[tp]-'a']=tv;
    // add leaf ts+1. It corresponds to transition through c.
    t[ts][c]=ts+1;l[ts+1]=ls;p[ts+1]=ts;
    // update info for the current node - remember to mark ts as parent of tv
    l[tp]=tp;p[tp]=ts;t[p[ts]][a[l[ts]]-'a']=ts;ts+=2;
    // prepare for descent
    // tp will mark where we are in the current suffix
    tv=s[p[ts-2]];tp=l[ts-2];
    // while the current suffix is not over, descend
    while (tp<r[ts-2]) {tv=t[tp][a[tp]-'a'];tp+=r[tp]-l[tp]+1;}
    // if we're in a node, add a suffix link to it, otherwise add the link to ts
    // (we'll create ts on next iteration).
    if (tp==r[ts-2]+1) s[ts-2]=tv; else s[ts-2]=ts;
    // add tp to the new edge and return to add letter to suffix
    tp=r[tp]-(tp-r[ts-2])+2;goto suff;
}
}

void build() {
    ts=2;
    tv=0;
    tp=0;
    fill(r,r+N,(int)a.size()-1);
    // initialize data for the root of the tree
    s[0]=1;
    l[0]=-1;
    r[0]=-1;
    l[1]=-1;
    r[1]=-1;
    memset(t,-1,sizeof t);
    fill(t[1],t[1]+26,0);
    // add the text to the tree, letter by letter
    for (la=0; la<(int)a.size(); ++la)
        ukkadd(a[la]-'a');
}
Practice

```

5.13 Z-Function

```

vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min(r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}

```

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;

```



```
String removed_period = sentence.substr(0, sentence.length()-1).trim();
if (pattern.matcher (removed_period).find()){
    System.out.println ("Good");
} else {
    System.out.println ("Bad!");
}
}
}
```

6.4 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 999999997.
// The largest prime smaller than 100000000000 is 99999999989.
// The largest prime smaller than 1000000000000 is 999999999971.
// The largest prime smaller than 10000000000000 is 9999999999973.
// The largest prime smaller than 100000000000000 is 9999999999989.
// The largest prime smaller than 1000000000000000 is 99999999999937.
// The largest prime smaller than 10000000000000000 is 99999999999997.
// The largest prime smaller than 100000000000000000 is 999999999999989.
// The largest prime smaller than 1000000000000000000 is 999999999999997.
// The largest prime smaller than 10000000000000000000 is 9999999999999989.
```

6.5 C++ input/output

```
#include <iostream>
#include <iomanip>
#include <bits/stdc++.h>

using namespace std;

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    srand((unsigned int)time(NULL));

    // Output 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;
}
```

6.6 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;
}
```

6.7 Latitude/longitude

```
/*
Converts from rectangular coordinates to latitude/longitude and vice
versa. Uses degrees (not radians).
*/

#include <iostream>
#include <cmath>

using namespace std;

struct ll
{
    double r, lat, lon;
```

```

};

struct rect
{
    double x, y, z;
};

ll convert(rect& P)
{
    ll Q;
    Q.r = sqrt(P.x*P.x+P.y*P.y+P.z*P.z);
    Q.lat = 180/M_PI*asin(P.z/Q.r);
    Q.lon = 180/M_PI*acos(P.x/sqrt(P.x*P.x+P.y*P.y));

    return Q;
}

rect convert(ll& Q)
{
    rect P;
    P.x = Q.r*cos(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
    P.y = Q.r*sin(Q.lon*M_PI/180)*cos(Q.lat*M_PI/180);
    P.z = Q.r*sin(Q.lat*M_PI/180);

    return P;
}

int main()
{
    rect A;
    ll B;

    A.x = -1.0; A.y = 2.0; A.z = -3.0;

    B = convert(A);
    cout << B.r << " " << B.lat << " " << B.lon << endl;

    A = convert(B);
    cout << A.x << " " << A.y << " " << A.z << endl;
}

```

6.8 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 x 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)), B = A;
    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;
    }
}

```

6.9 SAT-2

```

int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;

void dfs1(int v) {
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i) {
        int to = g[v][i];
        if (!used[to])
            dfs1(to);
    }
    order.push_back(v);
}

void dfs2(int v, int cl) {
    comp[v] = cl;
    for (size_t i=0; i<gt[v].size(); ++i) {
        int to = gt[v][i];
        if (comp[to] == -1)
            dfs2(to, cl);
    }
}

int main() {
    //(a || b) to !a=>b and !b=>a. a is 2*i and !a is 2*i+1
    used.assign(n, false);
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs1(i);

    comp.assign(n, -1);
    for (int i=0, j=0; i<n; ++i) {
        int v = order[n-i-1];
        if (comp[v] == -1)
            dfs2(v, j++);
    }

    //Variable and its negative in different components=>contradiction
    for (int i=0; i<n; ++i)
        if (comp[i] == comp[i^1]) {
            puts("NO SOLUTION");
            return 0;
        }

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

```

```
    int ans = comp[i] > comp[i^1] ? i : i^1;
    printf ("%d ", ans);
}
```

6.10 Ternary Search

```
while (r - l > EPS) {
    double m1 = l + (r - l) / 3,
           m2 = r - (r - l) / 3;
    if (f (m1) < f (m2))
        l = m1;
    else
        r = m2;
}
//Unimodal/Convex max(g,f), g+f
```
