IST ACM-ICPC Notebook 2016-17

AncelISTas1911

November 18, 2016

Contents

1 Miscellaneous

1.1 Default code

```
#include <bits/stdc++.h>
#define _ ios_base::sync_with_stdio(0);cin.tie(0);
#define FOR(i,a,b) for (int i=(a);i<(b);i++)
#define SZ(x) ((int)(x).size())
using namespace std;
```

1.2 C++ input/output

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

using namespace std;

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

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

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

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

1.3 STL stuff

```
// Example for using stringstreams and next_permutation
#include <bits/stdc++.h>

using namespace std;

struct C{
    bool operator()(const int &a, const int &b) const{
        return a > b;
    }
};

int main(void){
  vector<int> v = {1, 2, 3, 6};

  // upper_bound and lower_bound

  cout << *upper_bound(v.begin(), v.end(), 3) << endl; // exactly
       greater -> result = 6
  cout << *lower_bound(v.begin(), v.end(), 3) << endl; // equal or
       greater -> result = 3

  auto u = upper_bound(v.begin(), v.end(), 6);
  cout << (u == v.end() ? "end" : to_string(*u)) << endl; // if greater
        not available -> result = end
  cout << *lower_bound(v.begin(), v.end(), 7) << endl; // if greater or
        equal not available -> result = end

  cout << *upper_bound(v.begin(), v.end(), 3, less<int>()) << endl; //
       exactly greater -> result = 6
  cout << *lower_bound(v.begin(), v.end(), 3, less<int>()) << endl; //
       equal or greater -> result = 3

  // string to int, int to string

  cout << stoi("345") << " " << atoi("345") << " " << to_string(345) <<
        endl;

  // Permutations

  // Expected output: 1 2 3 6
  //                  1 2 6 3
  //                  ...
  //                  6 3 2 1

  do {
    ostringstream oss;
    oss << v[0] << " " << v[1] << " " << v[2] << " " << v[3];

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

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

  v.clear();

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

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

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

  // Expected output: 1 2 3
  for (size_t i = 0; i < v.size(); i++)
```

```
    cout << v[i] << " ";
  cout << endl;

  // custom comparator

  set<int, C> s;
}
```

1.4 Priority Queue

```
// priority queue having minimum at top

#include <queue>,<functional>
priority_queue< T, vector<T>, greater<T> > pqueue;

// priority queue with custom comparing function
#include <queue>
struct cmp {
        bool operator ()(const int a, const int b) {
                return ((a)<(b));
        }
};
priority_queue<int, vector<int>, cmp> q;
```

1.5 Dates (Java)

```
// Example of using Java's built-in date calculation routines

import java.text.SimpleDateFormat;
import java.util.*;

public class Dates {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        SimpleDateFormat sdf = new SimpleDateFormat("M/d/yyyy");
        while (true) {
            int n = s.nextInt();
            if (n == 0) break;
            GregorianCalendar c = new GregorianCalendar(n, Calendar.
                  JANUARY, 1);
            while (c.get(Calendar.DAY_OF_WEEK) != Calendar.SATURDAY)
                c.add(Calendar.DAY_OF_YEAR, 1);
            for (int i = 0; i < 12; i++) {
                System.out.println(sdf.format(c.getTime()));
                while (c.get(Calendar.MONTH) == i) c.add(Calendar.
                      DAY_OF_YEAR, 7);
            }
        }
    }
}
```

2 Graph algorithms

2.1 Fast Dijkstra's algorithm - Stanford

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

#include <queue>
#include <cstdio>

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

int main() {

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

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

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

        printf("%d\n", dist[t]);
        if (dist[t] < INF)
                for (int i = t; i != -1; i = dad[i])
                        printf("%d%c", i, (i == s ? '\n' : ' '));
        return 0;
}

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

Expected:
5
4 2 3 0
*/
```

2.2 Eulerian path - Stanford

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

struct Edge
{
        int next_vertex;
        iter reverse_edge;

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

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

vector<int> path;

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

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

2.3 Bellman Ford (Shortest path with negative edges)

```
// Time Complexity: O(V * E)
// Input: ne, nv, src, end, edges[N] (list of edges)
// Output: Shortest path from src to every vertex on the graph (iff
      bellman_ford() returns 0. 1 means negative cycle)

#define N 1000
struct Edge {
        int u, v, w;
};
int nv, ne, src, d[N];
Edge edges[N];
int bellman_ford() {
        memset(d, 0x3f, sizeof(d));
        d[src] = 0;
        for (int i=0; i < nv-1; i++)
                for (int j=0; j < ne; j++)
                        if (d[edges[j].u] + edges[j].w < d[edges[j].v])
                                d[edges[j].v] = d[edges[j].u] + edges[j
                                        ].w;
        for (int i=0; i < ne; i++)
                if (d[edges[i].u] + edges[i].w < d[edges[i].v])
```

```
				return 1;
		return 0;
}
```

2.4 Floyd-Wrashall (All-pairs shortest path)

```
// Time Complexity: O(N^3)
// Input: cost (adjacency matrix with cost)
// Output: Shortest path between all pair of nodes

#include <algorithm>
#define FOR(i,a,b) for(int i=(a); i<(b); i++)
#define N 100
int path[N][N], cost[N][N];
void FloydWarshall() {
	FOR(i,0,n) FOR(j,0,n) path[i][j] = cost[i][j];
	FOR(k,0,n) FOR(i,0,n) FOR(j,0,n)
		path[i][j] = min(path[i][j], path[i][k]+path[k][j]);
}
```

2.5 Prim (MST)

```
//Complexidade: O(E log V)
//Dados iniciais: pair<distancia, vertice> na lista de adjacencia
//Dados finais:
//	d[v] -> distancia da aresta que liga a MST ao vertice v
//	parent[v] -> vertice a que esta ligado o vertice v
//	totalweight -> peso total da arvore

#include <vector>,<set>
#define NVERTICES 10010
vector< pair<int,int> > adjlist[NVERTICES];
set< pair<int,int> > heap;
int d[NVERTICES], parent[NVERTICES], totalweight;
void add(int cost, int v, int p) {
	if(cost<d[v]) {
		parent[v]=p;
		heap.erase(pair<int,int>(d[v], v));
		d[v]=cost;
		heap.insert(pair<int,int>(d[v], v));
	}
}

void prim(int root) {
	memset(d, 0x3f, sizeof(d)); // 0x3f3f3f3f > 1.000.000.000
	memset(parent, -1, sizeof(parent));
	totalweight=0;
	add(0, root, -1);
	while(!heap.empty()) {
		pair<int,int> cur = *heap.begin();
		totalweight+=d[cur.second];
		d[cur.second]=0; //vertex in MST
		heap.erase(heap.begin()); //pop closest vertex
		for(unsigned int i=0; i<adjlist[cur.second].size(); i
		      ++) //for each neighbour
			add(adjlist[cur.second][i].first, adjlist[cur.
			      second][i].second, cur.second); //add/
			      refresh distance
	}
}
```

2.6 Kruskal - Stanford

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

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

using namespace std;

typedef int T;

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

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

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

T Kruskal(vector <vector <T> >& w)
{
  int n = w.size();
  T weight = 0;

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

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

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

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

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

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

  return weight;
}

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

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

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

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

2.7 Maximum Bipartite Matching

```
// Time Complexity: O( V * E ) which at most is O(V^3)
// Input: adjency list graph graph[i] has all the nodes j to which node
      i can be connected
//Output:
//	- matchL[m] (right vertex to which left vertex m is matched, -1
      if not matched)
//	- matchR[n] (left vertex to which right vertex n is matched, -1
      if not matched)
//	- nmatches (number of matches)

#include <cstring>
#include <vector>
#define MAX 410
vector<int> graph[MAX];
bool seen[MAX];
int matchL[MAX], matchR[MAX], nmatches;
int nLeft, nRight;
bool findmatch(int leftv) {
	for(int i=0; i<(int)graph[leftv].size(); i++) {
		int rightv = graph[leftv][i];
		if (seen[rightv]) continue;
		seen[rightv]=true;
		if(matchR[rightv]==-1 || findmatch(matchR[rightv])) {
			nmatches += (matchR[rightv]==-1 ? 1:0);
			matchR[rightv]=leftv;
			matchL[leftv]=rightv;
			return true;
		}
	}
	return false;
}
void bpm() {
	memset(matchL, -1, sizeof(matchL));
	memset(matchR, -1, sizeof(matchR));
	memset(seen, 0, sizeof(seen));
	nmatches=0;
	for(int i=0; i<nLeft; i++) {
		findmatch(i);
		memset(seen, 0, sizeof(seen));
	}
}
```

2.8 Articulation Points

```
// Count articulation points of a graph
struct{ vector<int> edges;
        int dfs;
        int low; }typedef Node;
int n; Node graph[805]; bool vis[805];
int d; int best, count;

void dfs(int node){
    int i, neigh;
    vis[node] = true;
    graph[node].dfs = d++;
    graph[node].low = graph[node].dfs;
    for(i=0; i<(int)graph[node].edges.size(); i++){
        neigh = graph[node].edges[i];
        if(!vis[neigh]){
            dfs(neigh);
            graph[node].low = min(graph[node].low, graph[neigh].low);
            if(graph[node].dfs>1){
                if(graph[neigh].low >= graph[node].dfs && graph[node].
                    edges.size() > 1)
                    count++;
            }else{
                if(graph[neigh].dfs > 2){
                    count++;
                }
            }

        }else{
            graph[node].low = min(graph[node].low, graph[neigh].dfs);
        }
    }
}
int main(){
    int i, j, v;
    d=1;
    memset(vis, false, sizeof(vis));
    count=0;
    for(i=1; i<=n; i++)
        if(!vis[i])
            dfs(i);
    printf("%d\n", count);
    return 0;
}
```

2.9 Strongly Connected Components

```
// Time Complexity: O(V + E)
// Input: adjlist
// Output: set of SCC

#include <vector>,<stack>
#define N 100
struct NODE {
        int index, lowlink;
};
int n, ind;
NODE nodes[N];
stack<int> st;
bool instack[N];
vector<vector<int> > adjlist, SCC;
void connect(int v) {
        int w;
        nodes[v].index = nodes[v].lowlink = ind++;
        st.push(v);
        instack[v] = true;
        for (int i=0; i<SZ(adjlist[v]); i++) {
                w = adjlist[v][i];
                if (!nodes[w].index) {
                        connect(w);
                        nodes[v].lowlink = min(nodes[v].lowlink, nodes[
                            w].lowlink);
                }
                else if (instack[w])
                        nodes[v].lowlink = min(nodes[v].lowlink, nodes[
                            w].index);
        }
        if (nodes[v].lowlink == nodes[v].index) {
                vector<int> tmp;
                for(w = -1; w != v; ) {
                        w = st.top(); st.pop();
                        instack[w] = false;
                        tmp.push_back(w);
                }
                SCC.push_back(tmp);
        }
}
void tarjan() {
        ind = 1;
        for (int i=0; i<n; i++) if (!nodes[i].index) connect(i);
}
```

2.10 Strongly connected components - Stanford

```
#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
      ]);}
}
```

3 Flows

3.1 Ford-Fulkerson (Max Flow)

```
// Time Complexity: O(E * N_flow)
// Input: src, end, cap[i][j] (capacity between nodes i and j)
// Output: Maximum flow from src to end

#include <vector>
#define N 100
#define INF 1000000007
typedef vector< pair<int,int> > vii;
int n, cap[N][N], flow[N][N];
bool vis[N];

bool dfs(int src, int end, vii &path) {
        if (src == end) {
                path.push_back(make_pair(end, INF));
                return true;
        }
        vis[src] = true;
        for (int i=0; i < n; i++) {
                int res = cap[src][i] - flow[src][i];
                if (res > 0 && !vis[i]) {
                        path.push_back( make_pair(src, res) );
                        bool ret = dfs(i, end, path);
                        if (ret) { vis[src] = false; return true; }
                        path.pop_back();
                }
        }
        vis[src] = false;
        return false;
}
int max_flow(int src, int end) {
        vector< pair<int,int> > path;
        while (dfs(src, end, path)) {
                int val = INF;
                for (int i=0; i < (int)path.size(); i++)
                        val = min(val, path[i].second);
                for (int i=0; i < (int)path.size()-1; i++) {
                        int a=path[i].first, b=path[i+1].first;
                        flow[a][b] += val;
                        flow[b][a] -= val;
                }
                path.resize(0);
        }
        int ret = 0;
        for (int i=0; i < n; i++) ret += flow[src][i];
        return ret;
```

3.2 Edmonds-Karp (Max Flow)

```
// Time Complexity: O(V * E^2)
// Input: ne, nv, src, end, cap[i][j] (capacity between nodes i and j)
// Output: Maximum flow from src to end

#include <cstring>
#include <queue>
#define N 100
#define INF 1000000007
int n, cap[N][N], flow[N][N], pre[N], res[N];
void bfs(int src, int end) {
        queue<int> qu;
        memset(pre,-1,sizeof(pre));
```

```
        res[src] = INF; qu.push(src);
        while (!qu.empty() && pre[end] < 0) {
                int c = qu.front(); qu.pop();
                for (int i=0; i < n; i++) {
                        if (pre[i] < 0 && cap[c][i]-flow[c][i] > 0) {
                                qu.push(i);
                                pre[i] = c;
                                res[i] = min(res[c], cap[c][i]-flow[c][
                                        i]);
}       }       }       }
int max_flow(int src, int end) {
        if (src == end) return INF;
        while (true) {
                bfs(src, end);
                if (pre[end] < 0) break; // No more cap in pres
                for (int i=end; i != src; i=pre[i]) {
                        flow[pre[i]][i] += res[end];
                        flow[i][pre[i]] -= res[end];
                }
        }
        int ret = 0;
        for (int i=0; i < n; i++) ret += flow[src][i];
        return ret;
}
```

3.3 Min-cost max-flow - Stanford

```
// 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
```

3.4 Global min-cut - Stanford

```
// 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_cut = cut;
          best_weight = w[last];
        }
      } else {
        for (int j = 0; j < N; j++)
          w[j] += weights[last][j];
```

```
        added[last] = true;
      }
    }
  }
  return make_pair(best_weight, best_cut);
}

// 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
```

4 Data structures

4.1 Range Minimum Query

```
// Time Complexity: Query O(log N)
// Input:
//       N -> number of values in A
//       A[i] -> i-th value
//       M[i] -> minimum value position for
//               the interval assigned to the i-th node
//Output:        Minimum value in interval [i, j]

#define MAXN 1000
#include <cstdio>

int A[MAXN], M[MAXN], N;

void init(int node, int b, int e) {
        if (b == e) M[node] = b;
        else {
        //compute left and right subtrees ranges
                init(2*node, b, (b + e)/2);
                init(2*node + 1, (b + e)/2 + 1, e);
        //search for min value 1st, 2nd half of interval
                if (A[M[2 * node]] <= A[M[2 * node + 1]])
                        M[node] = M[2 * node];
                else
                        M[node] = M[2 * node + 1];
        }
}
// b and e are bounds of the current interval
// i and j are bounds of the query interval
int query(int node, int b, int e, int i, int j) {
        int p1, p2;

        //[b,e] doesn't intersect [i,j]
        if (i > e || j < b)             return -1;
        //[b,e] in [i,j]
        if (b >= i && e <= j)   return M[node];

        //compute the minimum position in the
        //left and right part of the interval
        p1 = query(2*node, b, (b + e)/2, i, j);
        p2 = query(2*node + 1, (b + e)/2 + 1, e, i, j);

        //overall minimum position
        if (p1 == -1)                           return M[node] = p2;
        if (p2 == -1)                           return M[node] = p1;
        if (A[p1] <= A[p2])             return M[node] = p1;
        return M[node] = p2;
}
```

4.2 Binary Indexed Tree - Stanford

```
#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;
}
```

4.3 KD-tree

```
// -----------------------------------------------------------------
// A straightforward, but probably sub-optimal KD-tree implmentation
// that's probably good enough for most things (current it's a
// 2D-tree)
//
//  - constructs from n points in O(n lg^2 n) time
//  - handles nearest-neighbor query in O(lg n) if points are well
//    distributed
//  - worst case for nearest-neighbor may be linear in pathological
//    case
//
// 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 kdnode
{
    bool leaf;      // true if this is a leaf node (has one point)
    point pt;       // the single point of this is a leaf
    bbox bound;     // bounding box for set of points in children

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

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

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

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

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

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

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

    // constructs a kd-tree from a points (copied here, as it sorts
        them)
    kdtree(const vector<point> &vp) {
        vector<point> v(vp.begin(), vp.end());
        root = new kdnode();
        root->construct(v);
    }
    ~kdtree() { delete root; }

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

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

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

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

//
    -------------------------------------------------------------------------

// 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;
}

//
    -------------------------------------------------------------------------
```

4.4 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);
  }
}
```

4.5 Lowest common ancestor

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

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

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

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

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

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

4.6 Fenwick Tree

```
class FenwickTree{
  vector<long long> v;
  int maxSize;

public:
  FenwickTree(int _maxSize) : maxSize(_maxSize+1) {
    v = vector<long long>(maxSize, 0LL);
  }

  void add(int where, long long what){
    for (where++; where <= maxSize; where += where & -where){
      v[where] += what;
    }
  }

  long long query(int where){
    long long sum = v[0];
    for (where++; where > 0; where -= where & -where){
      sum += v[where];
    }
    return sum;
  }

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

};
```

4.7 Segment Tree

```
class SegmentTree{
public:
  vector<int> arr, tree;
  int n;

  SegmentTree(){}
  SegmentTree(const vector<int> &arr) : arr(arr) {
    initialize();
  }

  //must be called after assigning a new arr.
  void initialize(){
    n = arr.size();
    tree.resize(4*n + 1);
```

```
    initialize(0, 0, n-1);
  }

  int query(int query_left, int query_right) const{
    return query(0, 0, n-1, query_left, query_right);
  }

  void update(int where, int what){
    update(0, 0, n-1, where, what);
  }

private:
  int initialize(int node, int node_left, int node_right);
  int query(int node, int node_left, int node_right,
            int query_left, int query_right) const;
  void update(int node, int node_left, int node_right,
              int where, int what);
};

int SegmentTree::initialize(int node,
                            int node_left, int node_right){
  if (node_left == node_right){
    tree[node] = node_left;
    return tree[node];
  }
  int half = (node_left + node_right) / 2;
  int ans_left = initialize(2*node+1, node_left, half);
  int ans_right = initialize(2*node+2, half+1, node_right);

  if (arr[ans_left] <= arr[ans_right]){
    tree[node] = ans_left;
  }else{
    tree[node] = ans_right;
  }
  return tree[node];
}

int SegmentTree::query(int node, int node_left, int node_right,
                       int query_left, int query_right) const{
  if (node_right < query_left || query_right < node_left)
    return -1;
  if (query_left <= node_left && node_right <= query_right)
    return tree[node];

  int half = (node_left + node_right) / 2;
  int ans_left = query(2*node+1, node_left, half,
                       query_left, query_right);
  int ans_right = query(2*node+2, half+1, node_right,
                        query_left, query_right);

  if (ans_left == -1) return ans_right;
  if (ans_right == -1) return ans_left;

  return(arr[ans_left] <= arr[ans_right] ? ans_left : ans_right);
}

void SegmentTree::update(int node, int node_left, int node_right,
                         int where, int what){
  if (where < node_left || node_right < where) return;
  if (node_left == where && where == node_right){
    arr[where] = what;
    tree[node] = where;
    return;
  }
  int half = (node_left + node_right) / 2;
  if (where <= half){
    update(2*node+1, node_left, half, where, what);
  }else{
    update(2*node+2, half+1, node_right, where, what);
  }
  if (arr[tree[2*node+1]] <= arr[tree[2*node+2]]){
    tree[node] = tree[2*node+1];
  }else{
    tree[node] = tree[2*node+2];
  }
}
```

5 Geometry

5.1 Convex hull - Stanford

```
// 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
```

5.2 Convex Hull

```
// Time Complexity: O(N log N)
// Input: vector<Point> P e H
// Output: H fica com pontos do convexhull elemento H[H.size - 1] = H
     [0]

void ConvexHull (vector<Point> P, vector<Point> & H) {
        int n = P.size(), k = 0;
        H.resize(2*n);
        // Sort points lexicographically
        sort(P.begin(), P.end()); /* use Point comparator*/
        // Build lower hull
```

```
        for (int i = 0; i < n; i++) {
                /* change to: "cross() < 0" to include colinears */
                while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
                H[k++] = P[i];
        }
        // Build upper hull
        for (int i = n-2, t = k+1; i >= 0; i--) {
                /* change to: "cross() < 0" to include colinears */
                while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
                H[k++] = P[i];
        }
        H.resize(k);
}
```

5.3 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+RotateCW90(a-b), c, c+RotateCW90(
       a-c));
}

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

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

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

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

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

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

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

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
```

```
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}

int main() {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  // area should be 5.0
  // centroid should be (1.1666666, 1.166666)
  PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
  vector<PT> p(pa, pa+4);
  PT c = ComputeCentroid(p);
  cerr << "Area: " << ComputeArea(p) << endl;
  cerr << "Centroid: " << c << endl;

  return 0;
}
```

5.4 Closest Pair

```
// O(N log N)

#include <cmath>,<cstdio>,<algorithm>,<set>
#define EPS 1e-7
using namespace std;

struct Point {
        double x , y;
        int index ;
        bool operator < ( const Point & a ) const {
                return y < a.y || (fabs(y-a.y) < EPS && x < a.x );
        }
} p[60010] , tmp;

int np;
set<Point> pontos;
set<Point>::iterator it ;

bool comparaX( const Point & a , const Point & b ) {
        return a.x < b.x ;
}
double distancia( const Point & a , const Point & b ) {
        double q = a.x - b.x , w = a.y - b.y;
        return sqrt( q*q + w*w );
}

int main() {
        int i , left , a1 , a2;
        scanf("%d\n", &np);                // input
        for (i = 0 ; i < np ; i++) {
                scanf("%lf %lf\n", &p[i].x , &p[i].y);
                p[i].index = i;
        }
        sort( p , p+np , comparaX ); // ordena pontos por coordenada X
        double dist = 2000000000.0 , d;                    // sweep line
        pontos.insert(p[0]);
        left = 0;
        for (i = 1 ; i < np ; i++) {                       // para cada
             ponto p[i]
                while (p[i].x - p[left].x > dist )       // remove todos
                      os pontos cuja distancia em X ao ponto actual (p[
                     i])
                        pontos.erase( p[left++] );       // e' maior ou
                               igual do que a menor distancia entre
                               pontos encontrada ate ao momento
                tmp.y = p[i].y - dist;
                it = pontos.lower_bound( tmp );
                while (it != pontos.end() && it->y < p[i].y+dist) {
                           // percorrer os pontos do set
                        d = distancia( p[i] , *it ); // com Y dentro do
                              intevalo
                        if ( d < dist ) {        // [ p[i].y - dist , p[
                               i].y + dist ]
                                dist = d;
                                a1 = it->index;
                                a2 = p[i].index;
                        }
                        it++;
                }
                pontos.insert(p[i]);
        }
        if (a1 > a2) // verifica a1 apareceu antes que a2 no input
                swap( a1 , a2 );
        printf("%d %d %.6f\n", a1, a2, dist);    // output
//      printf("%.4f\n", dist); // distancia calculada, debug only
        return 0;
}
```

6 Dynamic Programming

6.1 Longest increasing subsequence

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

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

using namespace std;

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

#define STRICTLY_INCREASNG

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

  for (int i = 0; i < v.size(); i++) {
#ifdef STRICTLY_INCREASNG
    PII item = make_pair(v[i], 0);
```

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

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

6.2 Longest common subsequence

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

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

using namespace std;

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

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

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

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

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

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

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

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

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

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

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

6.3 Partition Problem

```
// Input: A given arrangement S of non-negative numbers s1; ... ; sn
     and an integer k.
// Output: Partition S into k ranges, so to minimize the maximum sum
     over all the ranges.

int M[1000][100], D[1000][100];
void partition_i(vector<int> &v, int k) {
        int p[1000], i, n = v.size();
        v.insert(v.begin(),0);  p[0] = 0;
        for(i = 1;i < v.size(); i++)    p[i] = p[i - 1] + v[i];
        for (i = 1; i <= n; i++)                M[i][1]=p[i];
        for (i = 1; i <= k; i++)                M[1][i] = v[1];
        for (i = 2; i <= n; i++) {
                for (int j = 2; j<=k; j++) {
                        M[i][j] = INT_MAX<<1 - 1;
                        int s = 0;
                        for (int x = 1; x <= i - 1; x++) {
                                s = max(M[x][j-1], p[i] - p[x]);
                                if (M[i][j] > s)
                                        M[i][j] = s,    D[i][j] = x;
        }       }       }
        printf("%d\n", M[n][k]);
}
//n = number of elements of the initial set
void reconstruct_partition(
const vector<int> &S, int n, int k) {
        if (k == 1) {
                for (int i = 1; i <= n; i++) printf("%d ", S[i]);
                putchar('\n');
        } else {
                reconstruct_partition(S, D[n][k], k - 1);
                for (int i = D[n][k] + 1; i<= n; i++) printf("%d ", S[i
                    ]);
                putchar('\n');
        }
}
```

7 Math

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

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

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

using namespace std;

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

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

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

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

```cpp
// (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;
}
```

7.2 Fast Fourier transform

```cpp
// Convolution using the fast Fourier transform (FFT).
//
// INPUT:
//     a[1...n]
//     b[1...m]
//
// OUTPUT:
//     c[1...n+m-1] such that c[k] = sum_{i=0}^k a[i] b[k-i]
//
// Alternatively, you can use the DFT() routine directly, which will
// zero-pad your input to the next largest power of 2 and compute the
// DFT or inverse DFT.

#include <iostream>
#include <vector>
#include <complex>

using namespace std;

typedef long double DOUBLE;
typedef complex<DOUBLE> COMPLEX;
typedef vector<DOUBLE> VD;
typedef vector<COMPLEX> VC;

struct FFT {
  VC A;
  int n, L;

  int ReverseBits(int k) {
    int ret = 0;
    for (int i = 0; i < L; i++) {
      ret = (ret << 1) | (k & 1);
      k >>= 1;
    }
    return ret;
  }

  void BitReverseCopy(VC a) {
    for (n = 1, L = 0; n < a.size(); n <<= 1, L++) ;
    A.resize(n);
    for (int k = 0; k < n; k++)
      A[ReverseBits(k)] = a[k];
  }

  VC DFT(VC a, bool inverse) {
    BitReverseCopy(a);
    for (int s = 1; s <= L; s++) {
      int m = 1 << s;
      COMPLEX wm = exp(COMPLEX(0, 2.0 * M_PI / m));
      if (inverse) wm = COMPLEX(1, 0) / wm;
      for (int k = 0; k < n; k += m) {
        COMPLEX w = 1;
        for (int j = 0; j < m/2; j++) {
          COMPLEX t = w * A[k + j + m/2];
          COMPLEX u = A[k + j];
          A[k + j] = u + t;
          A[k + j + m/2] = u - t;
          w = w * wm;
        }
      }
    }
    if (inverse) for (int i = 0; i < n; i++) A[i] /= n;
    return A;
  }

  // c[k] = sum_{i=0}^k a[i] b[k-i]
  VD Convolution(VD a, VD b) {
    int L = 1;
    while ((1 << L) < a.size()) L++;
    while ((1 << L) < b.size()) L++;
    int n = 1 << (L+1);

    VC aa, bb;
    for (size_t i = 0; i < n; i++) aa.push_back(i < a.size() ? COMPLEX(
        a[i], 0) : 0);
    for (size_t i = 0; i < n; i++) bb.push_back(i < b.size() ? COMPLEX(
        b[i], 0) : 0);

    VC AA = DFT(aa, false);
    VC BB = DFT(bb, false);
    VC CC;
    for (size_t i = 0; i < AA.size(); i++) CC.push_back(AA[i] * BB[i]);
```

```
    VC cc = DFT(CC, true);

    VD c;
    for (int i = 0; i < a.size() + b.size() - 1; i++) c.push_back(cc[i
        ].real());
    return c;
  }

};

int main() {
  double a[] = {1, 3, 4, 5, 7};
  double b[] = {2, 4, 6};

  FFT fft;
  VD c = fft.Convolution(VD(a, a + 5), VD(b, b + 3));

  // expected output: 2 10 26 44 58 58 42
  for (int i = 0; i < c.size(); i++) cerr << c[i] << " ";
  cerr << endl;

  return 0;
}
```

8 Strings

8.1 Knuth-Morris-Prath (String matching)

```
// Time Complexity: O(len(W) + len(S))
// Input: S and W (W is the substring to search in S)
// Output: Position of the first match of W in S

#include <cstdlib>,<string>
int* compute_prefix(string w) {
        int m = w.length(), k = 0;
        int *pi = (int*)malloc(sizeof(int)*m);
        pi[0] = 0;
        for (int q=1; q<m; q++) {
                while (k > 0 && w[k] != w[q]) k = pi[k-1];
                if (w[k] == w[q]) k++;
                pi[q] = k;
        }
        return pi;
}
int kmp_match(string s, string w) {
        int *pi=compute_prefix(w);
        int q = 0, n = s.length(), m = w.length();
        for (int i=0; i<n; i++) {
                while (q > 0 && w[q] != s[i]) q = pi[q-1];
                if (w[q] == s[i]) q++;
                if (q == m) return i-m+1; // Match at pos i-m+1
        }
        return -1; // No Match
}
```

8.2 Suffix array - Stanford

```
// 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
  //                  2
  for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << endl;
  cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
// BEGIN CUT
#endif
// END CUT
```

8.3 Another Suffix array

```
// Complexity: O(n log n)

//Usage: Call SuffixArray::compute(s), where s is the
//       string you want the Suffix Array for.
//
// * * * IMPORTANT: The last character of s must compare less
//       than any other character (for example, do s = s + '\1';
//       before calling this function).

//Output:
// sa   = The suffix array. Contains the n suffixes of s sorted
//        in lexicographical order. Each suffix is represented
//        as a single integer (the position in the string
//        where it starts).
// rank = The inverse of the suffix array. rank[i] = the index
//        of the suffix s[i..n) in the pos array. (In other
//        words, sa[i] = k <==> rank[k] = i).
//        With this array, you can compare two suffixes in O(1):
//        Suffix s[i..n) is smaller than s[j..n) if and
//        only if rank[i] < rank[j].
// lcp  = The length of the longest common prefix between two
//        consecutive suffixes:
//        lcp[i] = lcp(s + sa[i], s + sa[i-1]). lcp[0] = 0.

namespace SuffixArray {
    int t, rank[MAXN], sa[MAXN], lcp[MAXN];

    bool compare(int i, int j){
        return rank[i + t] < rank[j + t];
```

```
    }

    void build(const string &s){
        int n = s.size();
        int bc[256];
        for (int i = 0; i < 256; ++i) bc[i] = 0;
        for (int i = 0; i < n; ++i) ++bc[s[i]];
        for (int i = 1; i < 256; ++i) bc[i] += bc[i-1];
        for (int i = 0; i < n; ++i) sa[--bc[s[i]]] = i;
        for (int i = 0; i < n; ++i) rank[i] = bc[s[i]];
        for (t = 1; t < n; t <<= 1){
            for (int i = 0, j = 1; j < n; i = j++){
                while (j < n && rank[sa[j]] == rank[sa[i]]) j++;
                if (j - i == 1) continue;
                int *start = sa + i, *end = sa + j;
                sort(start, end, compare);
                int first = rank[*start + t], num = i, k;
                for(; start < end; rank[*start++] = num){
                    k = rank[*start + t];
                    if (k != first and (i > first or k >= j))
                        first = k, num = start - sa;
                }
            }
        }
        // Remove this part if you don't need the LCP
        int size = 0, i, j;
        for(i = 0; i < n; i++) if (rank[i] > 0) {
            j = sa[rank[i] - 1];
            while(s[i + size] == s[j + size]) ++size;
            lcp[rank[i]] = size;
            if (size > 0) --size;
        }
        lcp[0] = 0;
    }
};

////////////////////////////////////////////////////////////////

// Applications:

// lcp(x,y) = min(lcp(x,x+1), lcp(x+1, x+2), ... , lcp(y-1, y))

void number_of_different_substrings(){
  // If you have the i-th smaller suffix, Si,
  //    it's length will be |Si| = n - sa[i]
  // Now, lcp[i] stores the number of
  //     common letters between Si and Si-1
  //     (s.substr(sa[i]) and s.substr(sa[i-1]))
  // so, you have |Si| - lcp[i] different strings
  //    from these two suffixes => n - lcp[i] - sa[i]
  for(int i = 0; i < n; ++i) ans += n - sa[i] - lcp[i];
}

void number_of_repeated_substrings(){
  // Number of substrings that appear at least twice in the text.
  // The trick is that all 'spare' substrings that can give us
  // Lcp(i - 1, i) can be obtained by Lcp(i - 2, i - 1)
  // due to the ordered nature of our array.
  // And the overall answer is
  // Lcp(0, 1) +
  //     Sum(max[0, Lcp(i, i - 1) - Lcp(i - 2, i - 1)])
  //     for 2 <= i < n
  // File Recover
  int cnt = lcp[1];
  for(int i=2; i < n; ++i){
    cnt += max(0, lcp[i] - lcp[i-1]);
  }
}


void repeated_m_times(int m){
  // Given a string s and an int m, find the size
  // of the biggest substring repeated m times (find the rightmost pos)
  // if a string is repeated m+1 times, then it's repeated m times too
  // The answer is the maximum, over i, of the longest common prefix
  // between suffix i+m-1 in the sorted array.
  int length = 0, position = -1, t;
  for (int i = 0; i <= n-m; ++i){
    if ((t = getLcp(i, i+m-1, n)) > length){
      length = t;
      position = sa[i];
    } else if (t == length) { position = max(position, sa[i]); }
  }
  // Here you'll get the rightmost position
  // (that means, the last time the substring appears)
  for (int i = 0; i < n; ){
    if (sa[i] + length > n) { ++i; continue; }
    int ps = 0, j = i+1;
    while (j <n && lcp[j] >= length){
      ps = max(ps, sa[j]);
      j++;
    }
    if(j - i >= m) position = max(position, ps);
    i = j;
  }
  if(length != 0)
    printf("%d %d\n", length, position);
  else
    puts("none");
}


void smallest_rotation(){
  // Reads a string of length k. Then just double it (s = s+s)
  // and find the suffix array.
  // The answer is the smallest i for which s.size() - sa[i] >= k
  // If you want the first appearence (and not the string)
  // you'll need the second cycle
  int best = 0;
  for (int i=0; i < n; ++i){
    if (n - sa[i] >= k){
      //Find the first appearence of the string
      while (n - sa[i] >= k){
        if(sa[i] < sa[best] && sa[i] != 0) best = i;
        i++;
      }
      break;
    }
  }
  if (sa[best] == k) puts("0");
  else printf("%d\n", sa[best]);
}
```

8.4 Aho Corasick

```
////////////////////////////////////////////////////////////////
//         Aho-Corasick's algorithm, as explained in         //
//          http://dx.doi.org/10.1145/360825.360855           //
////////////////////////////////////////////////////////////////

// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 6 * 50 + 10;

// Number of characters in the alphabet.
const int MAXC = 26;

// Output for each state, as a bitwise mask.
// Bit i in this mask is on if the keyword with index i
// appears when the machine enters this state.
int out[MAXS];

// Used internally in the algorithm.
int f[MAXS]; // Failure function
int g[MAXS][MAXC]; // Goto function, or -1 if fail.

// Builds the string matching machine.
//
// words - Vector of keywords. The index of each keyword is
//          important:
//          "out[state] & (1 << i)" is > 0 if we just found
//           word[i] in the text.
// lowestChar - The lowest char in the alphabet.
//               Defaults to 'a'.
// highestChar - The highest char in the alphabet.
//                Defaults to 'z'.
//                "highestChar - lowestChar" must be <= MAXC,
//                otherwise we will access the g matrix outside
//                its bounds and things will go wrong.
//
// Returns the number of states that the new machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(const vector<string> &words,
                         char lowestChar = 'a',
                         char highestChar = 'z') {
    memset(out, 0, sizeof out);
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);

    int states = 1; // Initially, we just have the 0 state

    for (int i = 0; i < words.size(); ++i) {
        const string &keyword = words[i];
        int currentState = 0;
        for (int j = 0; j < keyword.size(); ++j) {
            int c = keyword[j] - lowestChar;
            if (g[currentState][c] == -1) {
                // Allocate a new node
                g[currentState][c] = states++;
            }
            currentState = g[currentState][c];
        }
        // There's a match of keywords[i] at node currentState.
        out[currentState] |= (1 << i);
    }

    // State 0 should have an outgoing edge for all characters.
    for (int c = 0; c < MAXC; ++c) {
        if (g[0][c] == -1) {
            g[0][c] = 0;
        }
    }

    // Now, let's build the failure function
    queue<int> q;
    // Iterate over every possible input
    for (int c = 0; c <= highestChar - lowestChar; ++c) {
        // All nodes s of depth 1 have f[s] = 0
        if (g[0][c] != -1 and g[0][c] != 0) {
            f[g[0][c]] = 0;
            q.push(g[0][c]);
        }
    }
    while (q.size()) {
        int state = q.front();
        q.pop();
        for (int c = 0; c <= highestChar - lowestChar; ++c) {
            if (g[state][c] != -1) {
                int failure = f[state];
                while (g[failure][c] == -1) {
                    failure = f[failure];
                }
                failure = g[failure][c];
                f[g[state][c]] = failure;

                // Merge out values
                out[g[state][c]] |= out[failure];
```

```
                q.push(g[state][c]);
            }
        }
    }

    return states;
}

// Finds the next state the machine will transition to.
//
// currentState - The current state of the machine. Must be
//                between 0 and the number of states - 1,
//                inclusive.
// nextInput - The next character that enters into the machine.
//             Should be between lowestChar and highestChar,
//             inclusive.
// lowestChar - Should be the same lowestChar that was passed
//              to "buildMatchingMachine".

// Returns the next state the machine will transition to.
// This is an integer between 0 and the number of states - 1,
// inclusive.
int findNextState(int currentState, char nextInput,
                                    char lowestChar = 'a') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while (g[answer][c] == -1) answer = f[answer];
    return g[answer][c];
}


// How to use this algorithm:
//
// 1. Modify the MAXS and MAXC constants as appropriate.
// 2. Call buildMatchingMachine with the set of keywords to
//    search for.
// 3. Start at state 0. Call findNextState to incrementally
//    transition between states.
// 4. Check the out function to see if a keyword has been
//    matched.
//
// Example:
//
// Assume keywords is a vector that contains
// {"he", "she", "hers", "his"} and text is a string that
// contains "ahishers".
//
// Consider this program:
//
// buildMatchingMachine(keywords, 'a', 'z');
// int currentState = 0;
// for (int i = 0; i < text.size(); ++i) {
//    currentState = findNextState(currentState, text[i], 'a');
//
//    Nothing new, let's move on to the next character.
//    if (out[currentState] == 0) continue;
//
//    for (int j = 0; j < keywords.size(); ++j) {
//        if (out[currentState] & (1 << j)) {
//            //Matched keywords[j]
//            cout << "Keyword " << keywords[j]
//                 << " appears from "
//                 << i - keywords[j].size() + 1
//                 << " to " << i << endl;
//        }
//    }
// }
//
// The output of this program is:
//
// Keyword his appears from 1 to 3
// Keyword he appears from 4 to 5
// Keyword she appears from 3 to 5
// Keyword hers appears from 4 to 7

///////////////////////////////////////////////////////////////
//                End of Aho-Corasick's algorithm           //
///////////////////////////////////////////////////////////////
```

8.5 Dynamic Hashing

```
// N = size of the array. It is assumed that elements are
//     indexed from 1 to N, inclusive.
// B = the base for the hash. Must be > 0.
// P = The modulo for the hash. Must be > 0. Doesn't need
//     to be prime.
int N, B, P;

int tree[MAXN], base[MAXN];

void precomputeBases() {
    base[0] = 1;
    for (int i = 1; i <= N + 1; ++i) {
        base[i] = (1LL * base[i - 1] * B) % P;
    }
}

inline int mod(long long a) {
    int ans = a % P;
    if (ans < 0) ans += P;
    return ans;
}

// Usually you don't want to use this function directly,
// use 'put' below instead.
void add(int at, int what) {
    what = mod(what);
    int seen = 0;
    for (at++; at <= N + 1; at += at & -at) {
        tree[at] += (1LL * what * base[seen]) % P;
        tree[at] = mod(tree[at]);
        seen += at & -at;
    }
}

// Returns the hash for subarray [1..at].
int query(int at) {
    int ans = 0, seen = 0;
    for (at++; at > 0; at -= at & -at) {
        ans += (1LL * tree[at] * base[seen]) % P;
        ans = mod(ans);
        seen += at & -at;
    }
    return ans;
}

// Returns the hash for subarray [i..j]. That hash is:
//   a[i]*B^(j-i+1) + a[i+1]*B  (j-i) + a[i+2]*B^(j-i-1) + ...
// + a[j-2]*B^2 + a[j-1]*B^1 + a[j]*B^0 (mod P)
int hash(int i, int j) {
    assert(i <= j);
    int ans = query(j) - (1LL * query(i-1) * base[j-i+1]) % P;
    return mod(ans);
}

// Changes the number or char at position 'at' for 'what'.
void put(int at, int what) {
    add(at, -hash(at, at) + what);
}
```

8.6 Manacher

```
// Complejidad: O(n)
void manacher(const string &s) {
    int n = s.size();

    vector<int> d1(n);
    int l=0, r=-1;
    for (int i=0; i<n; ++i) {
        int k = (i>r ? 0 : min (d1[l+r-i], r-i)) + 1;
        while (i+k < n && i-k >= 0 && s[i+k] == s[i-k]) ++k;
        d1[i] = --k;
        if (i+k > r) l = i-k, r = i+k;
    }
    vector<int> d2(n);
    l=0, r=-1;
    for (int i=0; i<n; ++i) {
        int k = (i>r ? 0 : min (d2[l+r-i+1], r-i+1)) + 1;
        while (i+k-1 < n && i-k >= 0 && s[i+k-1] == s[i-k]) ++k;
        d2[i] = --k;
        if (i+k-1 > r) l = i-k, r = i+k-1;
    }

    // d1[i] = piso de la mitad de la longitud del pal ndromo
    //          impar m s largo cuyo centro es i.
    // d2[i] = mitad de la longitud del pal ndromo par m s
    //          largo cuyo centro de la derecha es i.

    for (int i = 0; i < n; ++i) {
        assert(is_palindrome( s.substr(i - d1[i], 2*d1[i] + 1) ));
        assert(is_palindrome( s.substr(i - d2[i], 2*d2[i])     ));

    }
}
```

8.7 Minimum Rotation

```
// Finds the lexicographically smallest rotation of string s.
// Returns the index that should be moved to the first position
// to achieve the smallest rotation.
// If there are two or more smallest rotations, returns the
// smallest index.
int minimum_rotation(string s) {
    int n = s.size();
    s = s + s;
    int mini = 0, p = 1, k = 0;
    while (p < n && mini + k + 1 < n) {
        if (s[mini + k] == s[p + k]) {
            k++;
        } else if (s[mini + k] < s[p + k]) {
            p = p + k + 1;
            k = 0;
        } else if (s[mini + k] > s[p + k]){
            mini = max(mini + k + 1, p);
            p = mini + 1;
            k = 0;
        }
    }
    // the actual minimum rotated string is  s.substr(mini, n)
    return mini;
}
```

8.8 Z algorithm

```
// Find z function
int n = s.size();
vector<int> z(n);
z[0] = 0;
for (int i = 1, l = 0, r = 0; i < n; ++i) {
    z[i] = 0;
    if (i <= r) z[i] = min(z[i - l], r - i + 1);
    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;
    }
}
```

9 Cool Stuff

9.1 Topological sort (C++)

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

#include <iostream>, <queue>, <cmath>, <vector>
using namespace std;

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

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

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

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

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

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

9.2 Union-find set - Stanford

```
#include <iostream>
#include <vector>
using namespace std;
int find(vector<int> &C, int x) { return (C[x] == x) ? x : C[x] = find(
     C, C[x]); }
void merge(vector<int> &C, int x, int y) { C[find(C, x)] = find(C, y);
     }
int main()
{
        int n = 5;
        vector<int> C(n);
        for (int i = 0; i < n; i++) C[i] = i;
        merge(C, 0, 2);
        merge(C, 1, 0);
        merge(C, 3, 4);
        for (int i = 0; i < n; i++) cout << i << " " << find(C, i) <<
             endl;
        return 0;
}
```

9.3 Miller-Rabin Primality Test (C)

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

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

typedef long long LL;

LL ModularMultiplication(LL a, LL b, LL m)
{
        LL ret=0, c=a;
        while(b)
        {
                if(b&1) ret=(ret+c)%m;
                b>>=1; c=(c+c)%m;
        }
        return ret;
}
LL ModularExponentiation(LL a, LL n, LL m)
{
        LL ret=1, c=a;
        while(n)
        {
                if(n&1) ret=ModularMultiplication(ret, c, m);
                n>>=1; c=ModularMultiplication(c, c, m);
        }
        return ret;
}
bool Witness(LL a, LL n)
{
        LL u=n-1;
  int t=0;
        while(!(u&1)){u>>=1; t++;}
        LL x0=ModularExponentiation(a, u, n), x1;
        for(int i=1;i<=t;i++)
        {
                x1=ModularMultiplication(x0, x0, n);
                if(x1==1 && x0!=1 && x0!=n-1) return true;
                x0=x1;
        }
        if(x0!=1) return true;
        return false;
}
LL Random(LL n)
{
  LL ret=rand(); ret*=32768;
        ret+=rand(); ret*=32768;
        ret+=rand(); ret*=32768;
        ret+=rand();
  return ret%n;
}
bool IsPrimeFast(LL n, int TRIAL)
{
  while(TRIAL--)
  {
    LL a=Random(n-2)+1;
    if(Witness(a, n)) return false;
  }
  return true;
}
```

9.4 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;
  }
}
```