ACM

From DJudgeWiki

Contents

Basics

Simple Combinatorial Algorithms

```
// Generate all permutations or all (order does matter) possible ways to choose M out of N.
// Results in N!/(N-M)! outputs (N! for M=N).
#include <cstdio>
#include <algorithm>

using namespace std;

int N, M;
char S[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

// gen(0) generates all choose(M out of N).
// replace M by N to get all N! permutations.
void gen(int index) {
  if (index == M) {
    // If done, output.
    for (int i = 0; i < M; ++i) {
      printf("%c", S[i]);
    }
    printf("\n");
  } else {
    // Else keep generating permutations.
    for (int i = index; i < N; ++i) {
      // By swapping index with { index, index + 1, ..., N - 1 }.
      // (Important to include itself).
      swap(S[index], S[i]);
      gen(index + 1);
      swap(S[index], S[i]);
    }
  }
}

int main() {
```

```
  scanf("%d%d", &N, &M);
  gen(0);
}
```

Graph Theory

Graph Representations

Two main ways to represent the graph in memory are:

- Adjacency matrix. int D[N][N].
- Adjacency lists. vector<int> G[N].

D[i][j] is the cost/length/other parameter of the edge going from i to j. Be careful with D[i][i], although it would usually have some physical meaning depending on the task.

G[i] lists the vertices j_1, j_2, j_M, such that the graph contains edges from i to j_1, from i to j_2, etc.

Adjacency matrix is easier to code, but requires more memory and, for some algorithms, increases their complexity.

Of course, sometimes graph edges are not stored directly but generated "on demand". The concepts of the algorithms below would, of course, still apply.

DFS

High-level code.

```
void dfs(int v) {
  if (!visited[v]) {
    visited[v] = true;
    for (each vertex u s.t. there is an edge "v -> u") {
      dfs(u);
    }
  }
}

int N;  // Number of vertexes.
void invoke_dfs() {
  for (int i = 0; i < N; ++i) {
    dfs(i);
  }
}
```

Basic implementations.

```
bool D[i][j];

void dfs_with adjacency_matrix(int v) {
  if (!visited[v]) {
    visited[v] = true;
    for (int u = 0; u < N; ++u) {
      if (D[v][u]){
        dfs(u);
      }
    }
  }
}

vector<vector<int> > G;

void dfs_with adjacency_lists(int v) {
  if (!visited[v]) {
    visited[v] = true;
    for (int u_index = 0; u_index < G[v].size(); ++u_index) {
      const int u = G[v][u_index];
      dfs(u);
    }
```

```
  }
}
```

The canonical form of DFS introduces the concept of DFS time. DFS time starts with one and ends with twice the number of vertexes. When vertex processing is starting, the vertex's "discovery time" is stated to be current DFS time, and DFS time is incremented. When vertex processing is done, the vertex's "finish time" is stated to be current DFS time, and DFS time is incremented.

DFS times, and discovery/finish time help solving multiple problems, namely topological sort, strongly connected components, finding biconnected components, bridges, and articulation points.

One important fact is that if we define discovery time of vertex V as "opening parenthesis of type V", and finish time of vertex V as "closing parenthesis of type V", then the string of 2*N parentheses, ordered by DFS time, will form a correct bracket sequence.

Of course, depending on the actual problem statement, you might need to store only discovery times or only finish times. In this case it's reasonable to enumerate them from 1 to N (or from 0 to N-1), instead of using even-only or odd-only numbers.

Code.

```
void dfs(int v) {
  assert(visited[v]);
  ++dfs_time;
  discovery_time[v] = dfs_time;
  for (each u where there is an edge [v, u]) {
    if (!visited[u]) {
      visited[u] = true;
      dfs(u);
    }
  }
  ++dfs_time;
  finish_time[v] = dfs_time;
}

int N;
void invoke_dfs() {
  visited = vector<bool>(N, false);
  for (int i = 0; i < N; ++i) {
    if (!visited[i]) {
      visited[i] = true;
      dfs(i);
    }
  }
}
```

Notice that in this implementation *visited* array is updated before the call to dfs() itself. This implies that visited [v] is now a precondition, which is assert-ed in the example code above.

Connected Components

Only applicable for undirected graphs.

Each run of DFS in invoke_dfs() functions above will cover one connected component. Change vector<int> visited by vector<int> color and add current_color parameter to make DFS color the vertexes according to their connected components.

To just verify whether undirectede graph is connected, check whether the first call to DFS visits all the vertexes.

```
// Pseudo-code, which, in fact, works.
visited = vector<bool>(N, false);
dfs(0);
if (visited == vector<bool>(N, true)) {
  // Graph is connected.
}
```

Topological Sort

Vertexes ordered by finish_time form the topological sort of the transposed graph. Reversed list of finish_time-s results in graph's topological sort.

```
// Accepted UVA 10305.
// Toplogical sort.

#include <cstdio>
#include <vector>

using namespace std;

// DFS to build topological sort.
void dfs(int index,
         const vector<vector<int> >& g,
         vector<bool>& visited,
         vector<int>& order) {
  // Lazy visited[index] updates.
  // Will re-enter the same vertex multiple times.
  if (!visited[index]) {
    visited[index] = true;
    // DFS all the other vertexes.
    for (int i = 0; i < g[index].size(); ++i) {
      dfs(g[index][i], g, visited, order);
    }
    // All the vertexes which are reachable from index
    // are already in the order, safe to add index now.
    order.push_back(index);
  }
}

int main() {
  int N, M;
  while (N = 0, M = 0, scanf("%d%d", &N, &M) == 2 && N > 0) {
    vector<vector<int> > G(N);
    for (int i = 0; i < M; ++i) {
      int a, b;
      scanf("%d%d", &a, &b);
      // The transposed graph is stored.
      // (Alternative would be to reverse topological sort before printing it).
      G[b-1].push_back(a-1);
    }
    vector<bool> visited(N, false);
    vector<int> order;
    // Run DFS.
    for (int i = 0; i < N; ++i) {
      dfs(i, G, visited, order);
    }
    // Print the topological sort.
    for (int i = 0; i < N; ++i) {
      if (i > 0) {
        printf(" ");
      }
      printf("%d", order[i] + 1);
    }
    printf("\n");
  }
}
```

Bridges

The bridge in undirected graph is the edge the removal of which increases the number of connected components by one.

```
// Accepted UVA "Critical Links".
// Warning! The code below should be altered for multigraphs
// (the graphs where multiple edges between the same pair of vertexes
// are possible).

#include <algorithm>
#include <cstdio>
#include <set>
#include <vector>
#include <cassert>

using namespace std;

// First DFS: Build DFS tree, mark banned edges, create the order.
```

```
void dfs1(int index,
          const vector<vector<int> >& g,
          vector<bool>& visited,
          vector<set<int> >& banned_edges,
          vector<int>& order) {
  assert(visited[index]);
  order.push_back(index);
  for (int i = 0; i < g[index].size(); ++i) {
    if (!visited[g[index][i]]) {
      banned_edges[index].insert(g[index][i]);
      visited[g[index][i]] = true;
      dfs1(g[index][i], g, visited, banned_edges, order);
    }
  }
}

// Second DFS: Color biconnected components.
void dfs2(int index,
          const vector<vector<int> >& g,
          vector<int>& component,
          int component_index,
          const vector<set<int> >& banned_edges) {
  for (int i = 0; i < g[index].size(); ++i) {
    if (component[g[index][i]] == 0 && !banned_edges[index].count(g[index][i])) {
      component[g[index][i]] = component_index;
      dfs2(g[index][i], g, component, component_index, banned_edges);
    }
  }
}

int main() {
  // Create the graph carefully to ban multiedges.
  int N;
  while (N = -1, scanf("%d", &N) == 1 && N >= 0) {
    set<pair<int, int> > edges;
    for (int i = 0; i < N; ++i) {
      int index, count, tmp;
      scanf("%d (%d)", &index, &count);
      for (int j = 0; j < count; ++j) {
        scanf("%d", &tmp);
        edges.insert(make_pair(min(index, tmp), max(index, tmp)));
      }
    }
    vector<vector<int> > G(N);
    for (set<pair<int, int> >::const_iterator cit = edges.begin();
         cit != edges.end();
         ++cit) {
      G[cit->first].push_back(cit->second);
      G[cit->second].push_back(cit->first);
    }
    // Run first DFS.
    vector<bool> visited(N, false);
    vector<set<int> > banned_edges(N);
    vector<int> order;
    for (int i = 0; i < N; ++i) {
      if (!visited[i]) {
        visited[i] = true;
        dfs1(i, G, visited, banned_edges, order);
      }
    }
    assert(order.size() == N);
    // Run second DFS and color biconnected components.
    vector<int> component(N, 0);
    int component_index = 0;
    for (int i = 0; i < N; ++i) {
      if (!component[order[i]]) {
        ++component_index;
        component[order[i]] = component_index;
        dfs2(order[i], G, component, component_index, banned_edges);
      }
    }
    // Output critical edges.
    vector<pair<int, int> > output;
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < G[i].size(); ++j) {
        if (i < G[i][j] && component[i] != component[G[i][j]]) {
          output.push_back(make_pair(i, G[i][j]));
        }
      }
    }
    sort(output.begin(), output.end());
    printf("%d critical links\n", output.size());
    for (int i = 0; i < output.size(); ++i) {
      printf("%d - %d\n", output[i].first, output[i].second);
    }
    printf("\n");
```

```
  }
}
```

Articulation Points

The vertex is called articulation point, if the number of connected components increases if this vertex, along with the edges adjacent to it, is removed.

```
// Accepted "Network" on UVA.
// Articulation points detection.

#include <algorithm>
#include <cassert>
#include <cstdio>
#include <set>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

// DFS:
// 1) Keeps track of discovery time.
// 2) Keeps track of min. discovery time of the vertex
//    reachable from the subtree starting from vertex "i"
//    by following *exactly one* back link.
// 3) Keeps track of the number of children in DFS tree per vertex.
void dfs(int index,
         const vector<vector<int> >& g,
         int& dfs_current_time,
         vector<int>& dfs_discovery_time,
         vector<int>& dfs_tree_children_count,
         vector<int>& dfs_min_backlink_time,
         vector<bool>& is_articulation_point) {
  ++dfs_current_time;
  dfs_discovery_time[index] = dfs_current_time;
  for (int i = 0; i < g[index].size(); ++i) {
    if (!dfs_discovery_time[g[index][i]]) {
      ++dfs_tree_children_count[index];
      dfs(g[index][i],
          g,
          dfs_current_time,
          dfs_discovery_time,
          dfs_tree_children_count,
          dfs_min_backlink_time,
          is_articulation_point);
      dfs_min_backlink_time[index] =
        min(dfs_min_backlink_time[index], dfs_min_backlink_time[g[index][i]]);
      if (dfs_min_backlink_time[g[index][i]] >= dfs_discovery_time[index]) {
        // If for at least one subtree
        // min_backlink_time[child_index] is >= than discovery_time[index],
        // then this subtree will disconnect upon removing "index" vertex,
        // and, therefore, "index" vertex is an articulation point.
        is_articulation_point[index] = true;
      }
    } else {
      dfs_min_backlink_time[index] =
        min(dfs_min_backlink_time[index], dfs_discovery_time[g[index][i]]);
    }
  }
}

int main() {
  int N;
  while (N = 0, scanf("%d\n", &N) == 1 && N > 0) {
    // Read the graph carefully, remove multiedges, parse ugly input format.
    set<pair<int, int> > edges;
    static char tmp[1024 * 1024];
    bool repeat;
    do {
      repeat = false;
      fgets(tmp, sizeof(tmp), stdin);
      istringstream is(tmp);
      int from;
      is >> from;
      if (from > 0) {
        repeat = true;
        int to;
        while (is >> to) {
          edges.insert(make_pair(min(from, to), max(from, to)));
        }
      }
    } while (repeat);
```

```
    vector<vector<int> > G(N);
    for (set<pair<int, int> >::const_iterator cit = edges.begin();
         cit != edges.end();
         ++cit) {
      G[cit->first - 1].push_back(cit->second - 1);
      G[cit->second - 1].push_back(cit->first - 1);
    }
    // Run DFS.
    int dfs_current_time = 0;
    vector<int> dfs_discovery_time(N, 0);
    vector<int> dfs_tree_children_count(N, 0);
    vector<int> dfs_min_backlink_time(N, N + 1);
    vector<bool> is_articulation_point(N, false);
    for (int i = 0; i < N; ++i) {
      if (dfs_discovery_time[i] == 0) {
        dfs(i,
            G,
            dfs_current_time,
            dfs_discovery_time,
            dfs_tree_children_count,
            dfs_min_backlink_time,
            is_articulation_point);
        // Special case for the roots of DFS forest.
        // Important! Should overwrite is_articulation_point[i] here,
        // not just set it to "true" if the number of children is > 1.
        is_articulation_point[i] = (dfs_tree_children_count[i] > 1);
      }
    }
    assert(dfs_current_time == N);
    printf("%d\n", count(is_articulation_point.begin(),
                         is_articulation_point.end(),
                         true));
  }
}
```

Strongly Connected Components

The extension of connectivity concept for directed graphs. SCC-s may be identified using two DFS-es.

```
// Accepted UVA Trust Groups.
// Strongly Connected Components.

#include <cstdio>
#include <map>
#include <string>
#include <vector>

using namespace std;

string get_name() {
  static char s[1024];
  fgets(s, sizeof(s), stdin);
  string r = s;
  while (!r.empty() && isspace(*r.rbegin())) {
    r.resize(r.length() + 1);
  }
  return r;
}

// First DFS: Ran in original graph, saves DFS tree finish time order.
void dfs1(int index,
          const vector<vector<int> >& g,
          vector<bool>& visited,
          vector<int>& order) {
  if (!visited[index]) {
    visited[index] = true;
    for (int i = 0; i < g[index].size(); ++i) {
      dfs1(g[index][i], g, visited, order);
    }
    order.push_back(index);
  }
}

// Second DFS: Ran in transposed graph in the order of vertexes
// being the inverted order of finish time from DFS1, output
// strongly connected components.
// (Might want to pass extra parameter connected_component_index
// and use vector<int> connected_component instead of plain "visited").
void dfs2(int index,
          const vector<vector<int> >& gt,
          vector<bool>& visited) {
  if (!visited[index]) {
    visited[index] = true;
    for (int i = 0; i < gt[index].size(); ++i) {
```

```
      dfs2(gt[index][i], gt, visited);
    }
  }
}

int main() {
  int P, T;
  while (scanf("%d %d\n", &P, &T) == 2 && P > 0) {
    map<string, int> index;
    vector<vector<int> > G(P), GT(P);
    for (int i = 0; i < P; ++i) {
      index[get_name()] = i;
    }
    // Create both original and transposed graph at the same time.
    for (int i = 0; i < T; ++i) {
      const int from = index[get_name()];
      const int to = index[get_name()];
      G[from].push_back(to);
      GT[to].push_back(from);
    }
    // Run DFS1.
    vector<bool> visited1(P, false);
    vector<int> order;
    order.reserve(P);
    for (int i = 0; i < P; ++i) {
      dfs1(i, G, visited1, order);
    }
    // Run DFS2 and count the number of strongly connected components.
    vector<bool> visited2(P, false);
    int total = 0;
    for (int i = P - 1; i >= 0; --i) {
      if (!visited2[order[i]]) {
        ++total;
        dfs2(order[i], GT, visited2);
      }
    }
    printf("%d\n", total);
  }
}
```

BFS

BFS is another way to traverse the graph. Instead of going "further and further" until the "dead end" is reached, BFS processed vertexes in the order of how far they are from the origin if counting each edge as one step. The vertexes to process are therefore kept in the queue.

(And by just replacing queue with priority_queue we result in a nice Dijkstra implementation).

Maximum Flow

Maximum flow is usually coded by finding augmenting paths in the residual network. Use BFS, not DFS, when looking for an augmenting path.

```
// Accepted pothole on SPOJ.
// Maxflow.

#include <vector>
#include <cstring>
#include <queue>
#include <cstdio>

using namespace std;

const int MAXN = 200;
int N;
int G[MAXN][MAXN];

int augment() {
  // Run BFS in residual network and find an augmenting path.
  queue<int> Q;
  vector<bool> visited(N, false);
  vector<int> from(N, -1);
  visited[0] = true;
  Q.push(0);
  while (!Q.empty()) {
    const int i = Q.front();
    Q.pop();
    for (int j = 0; j < N; ++j) {
```

```
      if (G[i][j] > 0 && !visited[j]) {
        visited[j] = true;
        from[j] = i;
        Q.push(j);
      }
    }
  }
  if (from[N-1] == -1) {
    // If no path, no augment is possible.
    return 0;
  } else {
    // Found an augmenting path.
    // In this problem it will always have capacity one.
    // In real life, need to find its capacity here,
    // as min G[augmenting_path[i]][augmenting_path[i+1]].
    // Also note that when using doubles, make augmenting path
    // capacity real_augmenting_paty_capacity + EPSILON.
    // This would guarantee that "if (G[i][j] > 0)"
    // will never run into G[i][j] = +0.00000000.
    int i = N-1;
    while (i > 0) {
      --G[from[i]][i];
      ++G[i][from[i]];
      i = from[i];
    }
    // Return the capacity of augmenting path.
    return 1;
  }
}

int main() {
  int K;
  scanf("%d", &K);
  for (int k = 0; k < K; ++k) {
    // Parse the input.
    scanf("%d", &N);
    memset(G, 0, sizeof(G));
    for (int i = 0; i < N - 1; ++i) {
      int M;
      scanf("%d", &M);
      for (int m = 0; m < M; ++m) {
        int j;
        scanf("%d", &j);
        --j;
        // Capacity magic is only about this problem's input format.
        G[i][j] = (i == 0 || j == (N - 1)) ? 1 : 987654321;
      }
    }
    // Keep augmenting.
    int answer = 0, value;
    while ((value = augment()) > 0) {
      answer += value;
    }
    printf("%d\n", answer);
  }
}
```

Bipartite Matching

Kuhn's algorithm is usually used in the contests.

```
// Accepted "New Year Congratulations" from acm.mipt.ru.
// Please be extremely careful to not swap N and M indexes.
#include <cstdio>
#include <vector>

using namespace std;

const int MAX_N = 1001;

int N, K, C;
vector<int> W[MAX_N];

vector<bool> V;
int D[MAX_N];

bool dfs(int i) {
  if(!V[i]) {
    // Try all "bottom" vertexes ("i"-s) which were not visited before.
    V[i] = true;
    for(vector<int>::const_iterator it = W[i].begin(); it != W[i].end(); ++it) {
      const int j = *it;
      // If:
```

```
      // 1) this "top" vertex ("j") does not have a match, OR
      // 2) it does have a match but we can find a better match for it.
      if(D[j] == 0 || dfs(D[j])) {
        // Then an augmenting path was found. Match them and return true.
        // Re-matching of the vertexes involved is already done by this point
        // inside the recursive call stack to dfs(), which just returned true,
        // therefore only one "D[j] = i" extra rematch should be done.
        D[j] = i;
        return true;
      }
    }
  }
  // No improvement to current matching was found.
  return false;
}

int main() {
  scanf("%d%d%d", &N, &K, &C);
  for (int c = 0; c < C; ++c) {
    int a, b;
    scanf("%d%d", &a, &b);
    // Notice that the reverse order is kept,
    // W[x] holds "top" neighbours of x-th "bottom" vertex.
    W[b].push_back(a);
  }

  for(int i = 1; i <= K; ++i) {
    V = vector<bool>(K + 1, false);
    dfs(i);
  }

  int n = 0;
  for(int i = 1; i <= N; ++i) {
    if(D[i]) {
      ++n;
    }
  }

  printf("%d\n", n);

  for(int i = 1; i <= N; ++i) {
    if(D[i]) {
      // Reverse storage order allows to print the answer
      // in a simple way.
      printf("%d %d\n", i, D[i]);
    }
  }
}
```

Geometry

Use dot-products and cross-products only. This way you'll solve most problems w/o having to deal with doubles if the input is integer.

When using doubles, be extremely careful with epsilons. *Never* compare doubles, which are the result of some computation, as "a == b", and replace each "a > b" by "a > b + EPSILON" and each "a >= b" by "a > b - EPSILON".

EPSILON value: it's usually OK to use 1e-7 for double and 1e-9 for long double.

Convex Hull

Graham's algorithm is strongly advised to use in offline case. It's O(N * logN), proved to be asymptotically fastest, and uses only point sorting and CrossProduct.

```
// Accepted BSHEEP on SPOJ.
// Convex hull.
// NOTE ON EPSILONS: Require two places to insert epsilons.

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

```
using namespace std;

typedef pair<int, int> point;

int CrossProduct(const point& a, const point& b, const point& c) {
  return
    a.first * (b.second - c.second) +
    b.first * (c.second - a.second) +
    c.first * (a.second - b.second);
}

vector<point> GrahamConvexHull(const vector<point>& polygon) {
  // The condition on whether a vertex is in top
  // or bottom hull is EPSILON-TIGHT. This is because if the vertex
  // is on [p_left,p_right] the line, or *almost* on this line,
  // the right thing is to not consider it
  // rather than to try inserting it into both parts.
  // The condition on whether a vertex in either top or bottom hull
  // should be removed when a new vertex is added, is EPSILON-LOOSE.
  // The policy is "if in doubt, remove it".
  // This is because "in doubt" would mean that a new vertex
  // is continuing the segment started by the previous vertex,
  // and therefore the previous vertex is not needed.
  if (polygon.size() == 1) {
    // No convex hull required.
    return polygon;
  } else {
    // Sort the points left to right, down-to-up for equal X-s.
    // Warning: sorting also should be done carefully when the input points are doubles.
    // The sort order here is "by Y first, then by X for equal Y-s".
    // In order for this to work with doubles,
    // "equal Y-s" should be coded manually as fabs(Y2-Y1) < EPSILON.
    vector<point> sorted_polygon(polygon);
    sort(sorted_polygon.begin(), sorted_polygon.end());
    // Save the first and the last points.
    const point p_left = sorted_polygon.front();
    const point p_right = sorted_polygon.back();
    vector<point> upper, lower;
    // Start both upper and lower hulls with the leftmost point.
    upper.push_back(p_left);
    lower.push_back(p_left);
    // Iterate through all points of the polygon except for the leftmost one.
    for (int i = 1; i < sorted_polygon.size(); ++i) {
      // Check whether the point belongs to the upper or lower part.
      // The last point belongs to both parts.
      // Skip the points laying strictly on the [p_left, p_right] line.
      const int cp = CrossProduct(p_left, sorted_polygon[i], p_right);
      // EPSILON-TIGHT.
      if (i + 1 == sorted_polygon.size() || cp < 0 /* -EPSILON */) {
        // In the upper hull. Relax upper hull and add this point.
        // EPSILON-LOOSE.
        while (upper.size() >= 2 &&
               CrossProduct(upper[upper.size() - 2],
                            upper[upper.size() - 1],
                            sorted_polygon[i]) >= 0 /* > -EPSILON */) {
          upper.pop_back();
        }
        upper.push_back(sorted_polygon[i]);
      }
      // EPSILON-TIGHT.
      if (i + 1 == sorted_polygon.size() || cp > 0 /* +EPSILON */) {
        // In the lower hull. Relax lower hull and add this point.
        // EPSILON-LOOSE.
        while (lower.size() >= 2 &&
               CrossProduct(lower[lower.size() - 2],
                            lower[lower.size() - 1],
                            sorted_polygon[i]) <= 0 /* < +EPSILON */) {
          lower.pop_back();
        }
        lower.push_back(sorted_polygon[i]);
      }
    }
    // Store the result, direct upper hull followed by reversed lower hull.
    vector<point> result;
    for (int i = 0; i < upper.size(); ++i) {
      result.push_back(upper[i]);
    }
    for (int i = lower.size() - 2; i > 0; --i) {
      result.push_back(lower[i]);
    }
    return result;
  }
}

double Length(const point& a, const point& b) {
  const int dx = b.first - a.first;
  const int dy = b.second - a.second;
  return sqrt(static_cast<double>(dx * dx + dy * dy));
```

```
}

int main() {
  int T;
  scanf("%d", &T);
  for (int t = 0; t < T; ++t) {
    int N;
    scanf("%d" ,&N);
    vector<point> P(N);
    for (int i = 0; i < N; ++i) {
      scanf("%d%d", &P[i].second, &P[i].first);
    }
    map<point, int> indexes;
    for (int i = N - 1; i >= 0; --i) {
      indexes[P[i]] = i + 1;
    }
    P = GrahamConvexHull(P);
    double perimeter = 0;
    for (int i = 0; i < P.size(); ++i) {
      perimeter += Length(P[i], P[(i + 1) % P.size()]);
    }
    printf("%.2lf\n", perimeter);
    if (!P.empty()) {
      for (int i = 0; i < P.size(); ++i) {
        if (i > 0) {
          printf(" ");
        }
        printf("%d", indexes[P[i]]);
      }
      printf("\n");
    }
    printf("\n");
  }
}
```

Disjoint Sets

Atef's implementation.

```
// Accepted Electrician on SGU.
// Disjoint Sets by Atef.

import java.util.*;

public class Electrician {
        public static void main(String[] args) {
                Scanner in = new Scanner(System.in);
                int n = in.nextInt();
                Wire[] a = new Wire[n];
                for (int i = 0; i < n; i++) {
                        a[i] = new Wire(i, in.nextInt(), in.nextInt(), in.nextInt(), in.nextInt());
                }
                int nodes = 0;
                HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
                for (int i = 0; i < a.length; i++) {
                        if (map.get(a[i].a) == null)
                                map.put(a[i].a, nodes++);
                        create(a[i].a = map.get(a[i].a));
                        if (map.get(a[i].b) == null)
                                map.put(a[i].b, nodes++);
                        create(a[i].b = map.get(a[i].b));
                }
                Arrays.sort(a);
                long sum = 0;
                for (int i = 0; i < a.length; i++) {
                        if (find(a[i].a) != find(a[i].b)) {
                                sum += a[i].p;
                                union(a[i].a, a[i].b);
                        }
                }
                StringBuffer sb = new StringBuffer();
                sb.append(sum + "\n");
                Arrays.sort(a, new Comparator<Wire>() {
                        public int compare(Wire o1, Wire o2) {
                                return o1.r == o2.r ? (int)(o1.p - o2.p) : o2.r - o1.r;
                        }
                });
                for (int i = 0; i < a.length; i++) {
                        sb.append((a[i].i + 1) + " ");
                }
                System.out.println(sb.toString().trim());
        }
```

```
        static int[] p = new int[31000], rank = new int[31000];
        static void create(int x) {
                p[x] = x;
                rank[x] = 0;
        }

        static void union(int x, int y) {
                int px = find(x);
                int py = find(y);
                if (rank[px] > rank[py])
                        p[py] = px;
                else
                        p[px] = py;
                if (rank[px] == rank[py])
                        rank[py] = rank[py] + 1;
        }

        static int find(int x) {
                if (x != p[x])
                        p[x] = find(p[x]);
                return p[x];
        }
}

class Wire implements Comparable<Wire> {
        int i, a, b, r;
        long p;

        public Wire(int i, int a, int b, int r, long p) {
                super();
                this.i = i;
                this.a = a;
                this.b = b;
                this.r = r;
                this.p = p;
        }

        public int compareTo(Wire w) {
                return r == w.r ? (int)(w.p - p) : w.r - r;
        }
}
```

KMP

```
// Substring matching in O(N+M) using Knuth-Morriss-Pratt algorithm.
// P is the "pattern" string (what is being searched for),
// T is the "test" string (in which the pattern is being searched).
void kmp(const string& t, const string& p) const {
  const int n = t.length();
  const int m = p.length();
  assert(m <= n);
  vector<int> b = kmp_preprocess(p);
  vector<int> result;
  int i = 0, j = 0;
  while (i < n) {
    while (!(j == -1 || t[i] == p[j])) {
      j = b[j];
    }
    ++i;
    ++j;
    if (j == m) {
      report_match(i - j);
      j = b[j];
    }
  }
  return result;
}

vector<int> kmp_preprocess(const string& p) {
  const int n = p.length();
  const int infinity = 987654321;
  vector<int> b(n + 1, infinity);
  int i = 0, j = -1;
  b[i] = j;
  while (i < n) {
    while (!(j == -1 || p[i] == p[j])) {
      j = b[j];
    }
    ++i;
    ++j;
    b[i] = j;
  }
```

```
  return b;
}
```

Ternary Search

```
// Accepted "Triathlon" on acm.timus.ru.
// Efficient ternary search (golden cut ratio implementation).

#include <cstdio>
#include <cmath>

const long double kEpsilon = 1e-8;
const long double kOne = 1;
const long double kInf = 1e200;

const int kIterations = 75;

long double kPhiLeft;
long double kPhiRight;

const int MAXN = 100;
int N, U[MAXN], V[MAXN], W[MAXN];
long double UV[MAXN], UW[MAXN], VW[MAXN], UVW[MAXN];

long double f(int i, long double k1, long double k2) {
  return ((kOne - k1) * VW[i] + k1 * (kOne - k2) * UW[i] + k1 * k2 * UV[i]) / UVW[i];
}

long double f3(int i0, long double k1, long double k2) {
  long double min_t = kInf;
  for (int i = 0; i < N; ++i) {
    if (i != i0) {
      const long double t = f(i, k1, k2);
      if (t < min_t) {
        min_t = t;
      }
    }
  }
  return min_t / f(i0, k1, k2);
}

long double f2(int i, long double k) {
  long double a = 0, b = 1;
  long double fc, fd;
  bool has_fc = false, has_fd = false;
  for (int iteration = 0; iteration < kIterations; ++iteration) {
    const long double c = a * kPhiRight + b * kPhiLeft;
    const long double d = a * kPhiLeft + b * kPhiRight;
    if (!has_fc) {
      fc = f3(i, k, c);
    }
    if (!has_fd) {
      fd = f3(i, k, d);
    }
    if (fd < fc) {
      has_fc = false;
      has_fd = true;
      fd = fc;
      b = d;
    } else {
      has_fc = true;
      has_fd = false;
      fc = fd;
      a = c;
    }
  }
  return f3(i, k, (a+b) / 2);
}

long double f1(int i) {
  long double a = 0, b = 1;
  long double fc, fd;
  bool has_fc = false, has_fd = false;
  for (int iteration = 0; iteration < kIterations; ++iteration) {
    const long double c = a * kPhiRight + b * kPhiLeft;
    const long double d = a * kPhiLeft + b * kPhiRight;
    if (!has_fc) {
      fc = f2(i, c);
    }
    if (!has_fd) {
      fd = f2(i, d);
    }
    if (fd < fc) {
```

```
      has_fc = false;
      has_fd = true;
      fd = fc;
      b = d;
    } else {
      has_fc = true;
      has_fd = false;
      fc = fd;
      a = c;
    }
  }
  return f2(i, (a+b) / 2);
}

bool solve(int i) {
  return f1(i) > 1.0 + kEpsilon;
}

int main() {
  kPhiRight = (sqrtl(5.0) - 1) / 2;
  kPhiLeft = 1.0 - kPhiRight;
  scanf("%d", &N);
  for (int i = 0; i < N; ++i) {
    scanf("%d%d%d", &U[i], &V[i], &W[i]);
    const long double u = U[i], v = V[i], w = W[i];
    UV[i] = u * v;
    UW[i] = u * w;
    VW[i] = v * w;
    UVW[i] = u * v * w;
  }
  for (int i = 0; i < N; ++i) {
    printf("%s\n", solve(i) ? "Yes" : "No");
  }
}
```