# Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

2. November 2014

## Inhaltsverzeichnis

# 1 Datenstrukturen

## 1.1 Union-Find

```
vector<int> parent, rank2; //mache compiler verbieten Variable mit Namen rank

int findSet(int n) { //Pfadkompression
  if (parent[n] != n) parent[n] = findSet(parent[n]);
  return parent[n];
}

void linkSets(int a, int b) { //union by rank
  if (rank2[a] < rank2[b]) parent[a] = b;
  else if (rank2[b] < rank2[a]) parent[b] = a;
  else {
    parent[a] = b;
    rank2[b]++;
  }
}

void unionSets(int a, int b) {
  if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
}
```

## 1.2 Segmentbaum)

```
int a[MAX_N], m[4 * MAX_N];

int query(int x, int y, int k = 0, int X = 0, int Y = MAX_N - 1) {
  if (x <= X && Y <= y) return m[k];
  if (y < X || Y < x) return -1000000000; //ein "neutrales" Element
  int M = (X + Y) / 2;
  return max(query(x, y, 2 * k + 1, X, M), query(x, y, 2 * k + 2, M + 1, Y));
}

void update(int i, int v, int k = 0, int X = 0, int Y = MAX_N - 1) {
  if (i < X || Y < i) return;
  if (X == Y) {
    m[k] = w;
    a[i] = w;
    return;
  }
  int M = (X + Y) / 2;
  update(i, v, 2 * k + 1, X, M);
  update(i, v, 2 * k + 2, M + 1, Y);
  m[k] = max(m[2 * k + 1], m[2 * k + 2]);
}

void init(int k = 0, int X = 0, int Y = MAX_N - 1) {
  if (X == Y) {
    m[k] = a[X];
    return;
  }
  int M = (X + Y) / 2;
  init(2 * k + 1, X, M);
  init(2 * k + 2, M + 1, Y);
  m[k] = max(m[2 * k + 1], m[2 * k + 2]);
}
```

# 2 Graph

## 2.1 Strongly Connected Components (TARJANS-Algorithmus)

```
int counter, sccCounter, n; //n == number of vertices
vector<bool> visited, inStack;
vector< vector<int> > adjlist;
vector<int> d, low, sccs;
stack<int> s;

```

```
void visit(int v) {
  visited[v] = true;
  d[v] = counter;
  low[v] = counter;
  counter++;
  inStack[v] = true;
  s.push(v);

  for (int i = 0; i < (int)adjlist[v].size(); i++) {
    int u = adjlist[v][i];
    if (!visited[u]) {
      visit(u);
      low[v] = min(low[v], low[u]);
    } else if (inStack[u]) {
      low[v] = min(low[v], low[u]);
    }
  }

  if (d[v] == low[v]) {
    int u;
    do {
      u = s.top();
      s.pop();
      inStack[u] = false;
      sccs[u] = sccCounter;
    } while(u != v);
    sccCounter++;
  }
}

void scc() {
  //read adjlist

  visited.clear(); visited.assign(n, false);
  d.clear(); d.resize(n);
  low.clear(); low.resize(n);
  inStack.clear(); inStack.assign(n, false);
  sccs.clear(); sccs.resize(n);

  counter = 0;
  sccCounter = 0;
  for (i = 0; i < n; i++) {
    if (!visited[i]) {
      visit(i);
    }
  }
  //sccs has the component for each vertex
}
```

## 2.2 Max-Flow (Edmonds-Karp-Algorithmus)

```
int s, t, f; //source, target, single flow
int res[MAX_V][MAX_V]; //adj-matrix
vector< vector<int> > adjList;
int p[MAX_V]; //bfs spanning tree

void augment(int v, int minEdge) {
  if (v == s) { f = minEdge; return; }
  else if (p[v] != -1) {
    augment(p[v], min(minEdge, res[p[v]][v]));
    res[p[v]][v] -= f; res[v][p[v]] += f;
}}

int maxFlow() { //first inititialize res, adjList, s and t
  int mf = 0;
  while (true) {
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;
    queue<int> q; q.push(s);
    memset(p, -1, sizeof(p));
    while (!q.empty()) { //BFS
```

```
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int j = 0; j < (int)adjList[u].size(); j++) {
        int v = adjList[u][j];
        if (res[u][v] > 0 && !vis[v]) {
          vis[v] = true; q.push(v); p[v] = u;
    }}}

    augment(t, INF); //add found path to max flow
    if (f == 0) break;
    mf += f;
  }
  return mf;
}
```

# 3 Geometry

## 3.1 Closest Pair

```
double squaredDist(point a, point b) {
  return (a.first-b.first) * (a.first-b.first) + (a.second-b.second) * (a.second-b.second);
}

bool compY(point a, point b) {
  if (a.second == b.second) return a.first < b.first;
  return a.second < b.second;
}

double shortestDist(vector<point> &points) {
  //check that points.size() > 1 and that ALL POINTS ARE DIFFERENT
  set<point, bool(*)(point, point)> status(compY);
  sort(points.begin(), points.end());
  double opt = 1e30, sqrtOpt = 1e15;
  auto left = points.begin(), right = points.begin();
  status.insert(*right); right++;

  while (right != points.end()) {
    if (fabs(left->first - right->first) >= sqrtOpt) {
      status.erase(*(left++));
    } else {
      auto lower = status.lower_bound(point(-1e20, right->second - sqrtOpt));
      auto upper = status.upper_bound(point(-1e20, right->second + sqrtOpt));
      while (lower != upper) {
        double cand = squaredDist(*right, *lower);
        if (cand < opt) {
          opt = cand;
          sqrtOpt = sqrt(opt);
        }
        ++lower;
      }
      status.insert(*(right++));
    }
  }
  return sqrtOpt;
}
```

# 4 Sonstiges

## 4.1 2-SAT

1. Bedingungen in 2-CNF formulieren.
2. Implikationsgraph bauen, $(a \vee b)$ wird zu $\neg a \Rightarrow b$ und $\neg b \Rightarrow a$.
3. Finde die starken Zusammenhangskomponenten.
4. Genau dann lösbar, wenn keine Variable mit ihrer Negation in einer SCC liegt.