Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

25. November 2014

Inhaltsverzeichnis

1 Datenstrukturen **2**
- 1.1 Union-Find 2
- 1.2 Segmentbaum 2
- 1.3 Range Minimum Query 2
- 1.4 STL-Tree 3

2 Graphen **3**
- 2.1 Minimale Spannbäume 3
- 2.2 Kürzeste Wege 3
  - 2.2.1 Algorithmus von Dijkstra 3
  - 2.2.2 Bellmann-Ford-Algorithmus 4
  - 2.2.3 Floyd-Warshall-Algorithmus . . . 4
- 2.3 Strongly Connected Components (Tarjans-Algorithmus) 4
- 2.4 Artikulationspunkte und Brücken 5
- 2.5 Eulertouren 6
- 2.6 Lowest Common Ancestor 7
- 2.7 Max-Flow (Edmonds-Karp-Algorithmus) . 7
  - 2.7.1 Maximum Edge Disjoint Paths . . . 8
  - 2.7.2 Maximum Independent Paths 8
  - 2.7.3 Maximal Cardinatlity Bipartite Mathcing 8

3 Geometrie **8**
- 3.1 Closest Pair 8
- 3.2 Geraden 9
- 3.3 Konvexe Hülle 9
- 3.4 Formeln - `std::complex` 10

4 Mathe **11**
- 4.1 ggT, kgV, erweiterter euklidischer Algorithmus 11
  - 4.1.1 Multiplikatives Inverses von x in $\mathbb{Z}/n\mathbb{Z}$ 12
- 4.2 Primzahlsieb von Eratosthenes 12
  - 4.2.1 Faktorisierung 12
  - 4.2.2 Mod-Exponent über $\mathbb{F}_p$ 13
- 4.3 LGS über $\mathbb{F}_p$ 13
- 4.4 Binomialkoeffizienten 13
- 4.5 Satz von Sprague-Grundy 13
- 4.6 Maximales Teilfeld 14

5 Strings **14**
- 5.1 Knuth-Morris-Pratt-Algorithmus 14
- 5.2 Trie 14
- 5.3 Suffix-Array 15
- 5.4 Longest Common Substring 15
- 5.5 Longest Common Subsequence 16

6 Java **16**
- 6.1 Introduction 16
- 6.2 BigInteger 17

7 Sonstiges **17**
- 7.1 2-SAT 17
- 7.2 Linear time sorting 17

1 Datenstrukturen

1.1 Union-Find

```
vector<int> parent, rank2; //manche 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] = v;
        a[i] = v;
        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]);
}
```

1.3 Range Minimum Query

```
vector<int> data(RMQ_SIZE);
vector<vector<int>> rmq(floor(log2(RMQ_SIZE)) + 1, vector<int>(RMQ_SIZE));

void initRMQ() {
```

```
    for(int i = 0, s = 1, ss = 1; s <= RMQ_SIZE; ss=s, s*=2, i++) {
        for(int l = 0; l + s <= RMQ_SIZE; l++) {
            if(i == 0) rmq[0][l] = l;
            else {
                int r = l + ss;
                rmq[i][l] = (data[rmq[i-1][l]] <= data[rmq[i-1][r]] ? rmq[i-1][l] : rmq[i-1][r]);
            }
        }
    }
}
//returns index of minimum! [a, b)
int queryRMQ(int l, int r) {
    if(l >= r) return l;
    int s = floor(log2(r-l)); r = r - (1 << s);
    return (data[rmq[s][l]] <= data[rmq[s][r]] ? rmq[s][l] : rmq[s][r]);
}
```

1.4 STL-Tree

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std; using namespace __gnu_pbds;
typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> Tree;
int main() {
    Tree X;
    for (int i = 1; i <= 16; i <<= 1) X.insert(i); // {1, 2, 4, 8, 16}
    cout << *X.find_by_order(3) << endl; // => 8
    cout << X.order_of_key(10) << endl; // => 4 = successor of 10 = min i such that X[i] >= 10
    return 0;
}
```

2 Graphen

2.1 Minimale Spannbäume

Benutze Algorithmus von Kruskal oder Algorithmus von Prim.

Schnitteigenschaft Für jeden Schnitt C im Graphen gilt: Gibt es eine Kante e, die echt leichter ist als alle anderen Schnittkanten, so gehört diese zu allen minimalen Spannbäumen. (⇒ Die leichteste Kante in einem Schnitt kann in einem minimalen Spannbaum verwendet werden.)

Kreiseigenschaft Für jeden Kreis K im Graphen gilt: Die schwerste Kante auf dem Kreis ist nicht Teil des minimalen Spannbaums.

2.2 Kürzeste Wege

2.2.1 Algorithmus von Dijkstra

Kürzeste Pfade in Graphen ohne negative Kanten.

```
priority_queue<ii, vector<ii>, greater<ii> > pq;
vector<int> dist;
dist.assign(NUM_VERTICES, INF);
dist[0] = 0;
pq.push(ii(0, 0));

while (!pq.empty()) {
    ii front = pq.top(); pq.pop();
    int curNode = front.second, curDist = front.first;

    if (curDist > dist[curNode]) continue;

    for (int i = 0; i < (int)adjlist[curNode].size(); i++) {
        int nextNode = adjlist[curNode][i].first, nextDist = curDist + adjlist[curNode][i].second;

```

```
        if (nextDist < dist[nextNode]) {
            dist[nextNode] = nextDist; pq.push(ii(nextDist, nextNode));
        }
    }
}
```

2.2.2 Bellmann-Ford-Algorithmus

Kürzestes Pfade in Graphen mit negativen Kanten. Erkennt negative Zyklen.

```
//n = number of vertices, edges is vector of edges
dist.assign(n, INF); dist[0] = 0;
parent.assign(n, -1);
for (i = 0; i < n - 1; i++) {
    for (j = 0; j < (int)edges.size(); j++) {
        if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
            dist[edges[j].to] = dist[edges[j].from] + edges[j].cost;
            parent[edges[j].to] = edges[j].from;
        }
    }
}
//now dist and parent are correct shortest paths
//next lines check for negative cycles
for (j = 0; j < (int)edges.size(); j++) {
    if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
        //NEGATIVE CYCLE found
    }
}
```

2.2.3 Floyd-Warshall-Algorithmus

Alle kürzesten Pfade im Graphen.

```
//initialize adjmat, adjmat[i][i] = 0, adjmat[i][j] = INF if no edge is between i and j, length otherwise
for (k = 0; k < MAX_V; k++) {
    for (i = 0; i < MAX_V; i++) {
        for (j = 0; j < MAX_V; j++) {
            if (adjmat[i][k] + adjmat[k][j] < adjmat[i][j]) adjmat[i][j] = adjmat[i][k] + adjmat[k][j];
        }
    }
}
```

2.3 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
}
```

2.4 Artikulationspunkte und Brücken

```
vector< vector<int> > adjlist;
vector<int> low;
vector<int> d;
vector<bool> isArtPoint;
vector< vector<int> > bridges; //nur fuer Bruecken
int counter = 0;

void visit(int v, int parent) {
    d[v] = low[v] = ++counter;
    int numVisits = 0, maxlow = 0;

    for (vector<int>::iterator vit = adjlist[v].begin(); vit != adjlist[v].end(); vit++) {
        if (d[*vit] == 0) {
            numVisits++;
            visit(*vit, v);
            if (low[*vit] > maxlow) {
                maxlow = low[*vit];
            }

            if (low[*vit] > d[v]) { //nur fuer Bruecken
                bridges[v].push_back(*vit); bridges[*vit].push_back(v);
            }

            low[v] = min(low[v], low[*vit]);
        } else {
            if (d[*vit] < low[v]) {
                low[v] = d[*vit];
            }
        }
    }

    if (parent == -1) {
        if (numVisits > 1) isArtPoint[v] = true;
    } else {
```

```
        if (maxlow >= d[v]) isArtPoint[v] = true;
    }
}

void findArticulationPoints() {
    low.clear(); low.resize(adjlist.size());
    d.clear(); d.assign(adjlist.size(), 0);
    isArtPoint.clear(); isArtPoint.assign(adjlist.size(), false);
    bridges.clear(); isBridge.resize(adjlist.size()); //nur fuer Bruecken
    for (int v = 0; v < (int)adjlist.size(); v++) {
        if (d[v] == 0) visit(v, -1);
    }
}
```

2.5 Eulertouren

- Zyklus existiert, wenn jeder Knoten geraden Grad hat (ungerichtet), bzw. bei jedem Knoten Ein- und Ausgangsgrad übereinstimmen (gerichtet).
- Pfad existiert, wenn alle bis auf (maximal) zwei Knoten geraden Grad haben (ungerichtet), bzw. bei allen Knoten bis auf zwei Ein- und Ausgangsgrad übereinstimmen, wobei einer eine Ausgangskante mehr hat (Startknoten) und einer eine Eingangskante mehr hat (Endknoten).
- **Je nach Aufgabenstellung überprüfen, wie isolierte Punkte interpretiert werden sollen.**
- Der Code unten läuft in Linearzeit. Wenn das nicht notwenidg ist (oder bestimmte Sortierungen verlangt werden), gehts mit einem `set` einfacher.

```
VISIT(v):
    forall e=(v,w) in E
    delete e from E
    VISIT(w)
    print e
```

Abbildung 1: Idee für Eulerzyklen

```
vector< vector<int> > adjlist;
vector< vector<int> > otherIdx;
vector<int> cycle;
vector<int> validIdx;

void swapEdges(int n, int a, int b) { // Vertauscht Kanten mit Indizes a und b von Knoten n.
    int neighA = adjlist[n][a];
    int neighB = adjlist[n][b];
    int idxNeighA = otherIdx[n][a];
    int idxNeighB = otherIdx[n][b];
    swap(adjlist[n][a], adjlist[n][b]);
    swap(otherIdx[n][a], otherIdx[n][b]);
    otherIdx[neighA][idxNeighA] = b;
    otherIdx[neighB][idxNeighB] = a;
}

void removeEdge(int n, int i) { // Entfernt Kante i von Knoten n (und die zugehoerige Rueckwaertskante).
    int other = adjlist[n][i];
    if (other == n) { //Schlingen
        validIdx[n]++;
        return;
    }
    int otherIndex = otherIdx[n][i];
    validIdx[n]++;
    if (otherIndex != validIdx[other]) {
        swapEdges(other, otherIndex, validIdx[other]);
    }
    validIdx[other]++;
}

//findet Eulerzyklus an Knoten n startend
//teste vorher, dass Graph zusammenhaengend ist! (isolierte Punkte sind ok)
```

```
//teste vorher, ob Eulerzyklus ueberhaupt existiert!
void euler(int n) {
    while (validIdx[n] < (int)adjlist[n].size()) {
        int nn = adjlist[n][validIdx[n]];
        removeEdge(n, validIdx[n]);
        euler(nn);
    }
    cycle.push_back(n); //Zyklus am Ende in cycle
}
```

2.6 Lowest Common Ancestor

```
//RMQ muss hinzugefuegt werden!
vector<int> visited(2*MAX_N), first(MAX_N, 2*MAX_N), depth(2*MAX_N);
vector<vector<int>> graph(MAX_N);

void initLCA(int gi, int d, int &c) {
    visited[c] = gi, depth[c] = d, first[gi] = min(c, first[gi]), c++;
    for(int gn : graph[gi]) {
        initLCA(gn, d+1, c);
        visited[c] = gi, depth[c] = d, c++;
    }
}
//[a, b]
int getLCA(int a, int b) {
    return visited[queryRMQ(min(first[a], first[b]), max(first[a], first[b]))];
}
//=> int c = 0; initLCA(0,0,c); initRMQ(); done!
```

2.7 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 inititalize 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;
}
```

2.7.1 Maximum Edge Disjoint Paths

Finde die maximale Anzahl Pfade von s nach t, die keine Kante teilen.

1. Setze s als Quelle, t als Senke und die Kapazität jeder Kante auf 1.
2. Der maximale Fluss entspricht der unterschiedlichen Pfade ohne gemeinsame Kanten.

2.7.2 Maximum Independent Paths

Finde die maximale Anzahl Pfade von s nach t, die keinen Knoten teilen.

1. Setze s als Quelle, t als Senke und die Kapazität jeder Kante *und jedes Knotens* auf 1.
2. Der maximale Fluss entspricht der unterschiedlichen Pfade ohne gemeinsame Knoten.

2.7.3 Maximal Cardinatlity Bipartite Mathcing

```
vector< vector<int> > adjlist;
vector<int> pairs; //for every node, stores the matching node on the other side or -1
vector<bool> visited;

bool dfs(int i) {
    if (visited[i]) return false;
    visited[i] = true;
    for (vector<int>::iterator vit = adjlist[i].begin(); vit != adjlist[i].end(); vit++) {
        if (pairs[*vit] < 0 || dfs(pairs[*vit])) {
            pairs[*vit] = i; pairs[i] = *vit; return true;
        }
    }
    return false;
}

int kuhn(int n, int m) { // n = nodes on left side (numbered 0..n-1), m = nodes on the right side
    pairs.assign(n + m, -1);
    int ans = 0;
    for (int i = 0; i < n; i++) {
        visited.assign(n + m, false);
        ans += dfs(i);
    }
    return ans; //size of the MCBM
}
```

3 Geometrie

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;
}
```

3.2 Geraden

```
struct pt { //complex<double> does not work here, becuase we need to set pt.x and pt.y
    double x, y;
    pt() {};
    pt(double x, double y) : x(x), y(y) {};
};

struct line {
    double a, b, c; //a*x+b*y+c, b=0 <=> vertical line, b=1 <=> otherwise
};

line pointsToLine(pt p1, pt p2) {
    line l;
    if (fabs(p1.x - p2.x) < EPSILON) {
        l.a = 1; l.b = 0.0; l.c = -p1.x;
    } else {
        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
        l.b = 1.0;
        l.c = -(double)(l.a * p1.x) - p1.y;
    }
    return l;
}

bool areParallel(line l1, line l2) {
    return (fabs(l1.a - l2.a) < EPSILON) && (fabs(l1.b - l2.b) < EPSILON);
}

bool areSame(line l1, line l2) {
    return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPSILON);
}

bool areIntersect(line l1, line l2, pt &p) {
    if (areParallel(l1, l2)) return false;
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
    if (fabs(l1.b) > EPSILON) p.y = -(l1.a * p.x + l1.c);
    else p.y = -(l2.a * p.x + l2.c);
    return true;
}
```

3.3 Konvexe Hülle

```
struct point {
  double x, y;
  point(){} point(double x, double y) : x(x), y(y) {}
  bool operator <(const point &p) const {
    return x < p.x || (x == p.x && y < p.y);
  }
};

```

```
// 2D cross product.
// Return a positive value, if OAB makes a counter-clockwise turn,
// negative for clockwise turn, and zero if the points are collinear.
double cross(const point &O, const point &A, const point &B){
  double d = (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
  if (fabs(d) < 1e-9) return 0.0;
  return d;
}

// Returns a list of points on the convex hull in counter-clockwise order.
// Colinear points are not in the convex hull, if you want colinear points in the hull remove "=" in the CCW-
    Test
// Note: the last point in the returned list is the same as the first one.
vector<point> convexHull(vector<point> P){
  int n = P.size(), k = 0;
  vector<point> H(2*n);

  // Sort points lexicographically
  sort(P.begin(), P.end());

  // Build lower hull
  for (int i = 0; i < n; i++) {
    while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0.0) k--;
    H[k++] = P[i];
  }

  // Build upper hull
  for (int i = n-2, t = k+1; i >= 0; i--) {
    while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0.0) k--;
    H[k++] = P[i];
  }

  H.resize(k);
  return H;
}
```

3.4 Formeln - `std::complex`

```
//komplexe Zahlen als Darstellung fuer Punkte
typedef pt complex<double>;
//Winkel zwischen Punkt und x-Achse in [0, 2 * PI), Winkel zwischen a und b
double angle = arg (a), angle_a_b = arg (a - b);
//Punkt rotiert um Winkel theta
pt a_rotated = a * exp (pt (0, theta));
//Mittelpunkt des Dreiecks abc
pt centroid = (a + b + c) / 3;
//Skalarprodukt
double dot(pt a, pt b) {
    return real(conj(a) * b);
}
//Kreuzprodukt, 0, falls kollinear
double cross(pt a, pt b) {
    return imag(conj(a) * b);
}
//wenn Eckpunkte bekannt
double areaOfTriangle(pt a, pt b, pt c) {
    return abs(cross(b - a, c - a)) / 2.0;
}
//wenn Seitenlaengen bekannt
double areaOfTriangle(double a, double b, double c) {
    double s = (a + b + c) / 2;
    return sqrt(s * (s-a) * (s-b) * (s-c));
}
// Sind die Dreiecke a1, b1, c1, and a2, b2, c2 aehnlich?
// Erste Zeile testet Aehnlichkeit mit gleicher Orientierung,
// zweite Zeile testst Aehnlichkeit mit unterschiedlicher Orientierung
bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
    return (
        (b2 - a2) * (c1 - a1) == (b1 - a1) * (c2 - a2) ||
```

```
        (b2 - a2) * (conj (c1) - conj (a1)) == (conj (b1) - conj (a1)) * (c2 - a2)
    );
}
//Linksknick von a->b nach a->c
double ccw(pt a, pt b, pt c) {
    return cross(b - a, c - a); //<0 => falls Rechtsknick, 0 => kollinear, >0 => Linksknick
}
//Streckenschnitt, Strecken a-b und c-d
bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
    if (ccw(a, b, c) == 0 && ccw(a, b, d) == 0) { //kollinear
        double dist = abs(a - b);
        return (abs(a - c) <= dist && abs(b - c) <= dist) || (abs(a - d) <= dist && abs(b - d) <= dist);
    }
    return ccw(a, b, c) * ccw(a, b, d) <= 0 && ccw(c, d, a) * ccw(c, d, b) <= 0;
}
//Entfernung von p zu a-b
double distToLine(pt a, pt b, pt p) {
    return abs(cross(p - a, b - a)) / abs(b - a);
}
//liegt p auf a-b
bool pointOnLine(pt a, pt b, pt p) {
    return abs(distToLine(a, b, p)) < EPSILON;
}
//testet, ob d in der gleichen Ebene liegt wie a, b, und c
bool isCoplanar(pt a, pt b, pt c, pt d) {
    return (b - a) * (c - a) * (d - a) == 0;
}
//berechnet den Flaecheninhalt eines Polygons (nicht selbstschneidend)
double areaOfPolygon(vector<pt> &polygon) { //jeder Eckpunkt nur einmal im Vektor
    double res = 0; int n = polygon.size();
    for (int i = 0; i < (int)polygon.size(); i++)
        res += real(polygon[i]) * imag(polygon[(i + 1) % n]) - real(polygon[(i + 1) % n]) * imag(polygon[i]);
    return 0.5 * abs(res);
}
//testet, ob sich zwei Rechtecke (p1, p2) und (p3, p4) schneiden (jeweils gegenueberliegende Ecken)
bool rectIntersection(pt p1, pt p2, pt p3, pt p4) {
    double minx12 = min(real(p1), real(p2)), maxx12 = max(real(p1), real(p2));
    double minx34 = min(real(p3), real(p4)), maxx34 = max(real(p3), real(p4));
    double miny12 = min(imag(p1), imag(p2)), maxy12 = max(imag(p1), imag(p2));
    double miny34 = min(imag(p3), imag(p4)), maxy34 = max(imag(p3), imag(p4));
    return (maxx12 >= minx34) && (maxx34 >= minx12) && (maxy12 >= miny34) && (maxy34 >= miny12);
}
//testet, ob ein Punkt im Polygon liegt (beliebige Polygone)
bool pointInPolygon(pt p, vector<pt> &polygon) { //jeder Eckpunkt nur einmal im Vektor
    pt rayEnd = p + pt(1, 1000000);
    int counter = 0, n = polygon.size();
    for (int i = 0; i < n; i++) {
        pt start = polygon[i], end = polygon[(i + 1) % n];
        if (lineSegmentIntersection(p, rayEnd, start, end)) counter++;
    }
    return counter & 1;
}
```

4 Mathe

4.1 ggT, kgV, erweiterter euklidischer Algorithmus

```
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd (b, a % b);
}

ll lcm(ll a, ll b) {
    return a * (b / gcd(a, b)); //Klammern gegen Overflow
}
```

```
//Accepted in Aufgabe mit Forderung: |X|+|Y| minimal (primaer) und X<=Y (sekundaer)
//hab aber keinen Beweis dafuer :)
```

```
ll x, y, d; //a * x + b * y = d = ggT(a,b)
void extendedEuclid(ll a, ll b) {
    if (!b) {
        x = 1; y = 0; d = a; return;
    }
    extendedEuclid(b, a % b);
    ll x1 = y; ll y1 = x - (a / b) * y;
    x = x1; y = y1;
}
```

4.1.1 Multiplikatives Inverses von x in $\mathbb{Z}/n\mathbb{Z}$

Sei $0 \leq x < n$. Definiere $d := gcd(x, n)$.
Falls $d = 1$:

- Erweiterter euklidischer Algorithmus liefert α und β mit $\alpha x + \beta n = 1$
- Nach Kongruenz gilt $\alpha x + \beta n \equiv \alpha x \equiv 1 \mod n$
- $x^{-1} :\equiv \alpha \mod n$

Falls $d \neq 1$: es existiert kein x^{-1}

```
ll multInv(ll n, ll p) { //berechnet das multiplikative Inverse von n in F_p
    extendedEuclid(n, p); //implementierung von oben
    x += ((x / p) + 1) * p;
    return x % p;
}
```

4.2 Primzahlsieb von Eratosthenes

```
vector<int> primes;
void primeSieve(ll n) { //berechnet die Primzahlen kleiner n
    vector<int> isPrime(n,true);
    for(ll i = 2; i < n; i+=2) {
        if(isPrime[i]) {
            primes.push_back(i);
            if(i*i <= n) {
                for(ll j = i; i*j < n; j+=2) isPrime[i*j] = false;
            }
        }
        if(i == 2) i--;
    }
}
```

4.2.1 Faktorisierung

```
const ll PRIME_SIZE = 10000000;
vector<int> primes; //call primeSieve(PRIME_SIZE); before

//Factorize the number n
vector<int> factorize(ll n) {
    vector<int> factor;
    ll num = n;
    int pos = 0;
    while(num != 1)  {
        if(num % primes[pos] == 0) {
            num /= primes[pos];
            factor.push_back(primes[pos]);
        }
        else pos++;
        if(primes[pos]*primes[pos] > n) break;
    }
    if(num != 1) factor.push_back(num);
    return factor;
}
```

4.2.2 Mod-Exponent über $\mathbb{F}_p$

```
ll modPow(ll b, ll e, ll p) {
    if (e == 0) return 1;
    if (e == 1) return b;
    ll half = modPow(b, e / 2, p), res = (half * half) % p;
    if (e & 1) res *= b; res %= p;
    return res;
}
```

4.3 LGS über $\mathbb{F}_p$

```
void normalLine(ll n, ll line, ll p) { //normalisiert Zeile line
    ll factor = multInv(mat[line][line], p); //Implementierung von oben
    for (ll i = 0; i <= n; i++) {
        mat[line][i] *= factor;
        mat[line][i] %= p;
    }
}

void takeAll(ll n, ll line, ll p) { //zieht Vielfaches von line von allen anderen Zeilen ab
    for (ll i = 0; i < n; i++) {
        if (i == line) continue;
        ll diff = mat[i][line]; //abziehen
        for (ll j = 0; j <= n; j++) {
            mat[i][j] -= (diff * mat[line][j]) % p;
            while (mat[i][j] < 0) {
                mat[i][j] += p;
            }
        }
    }
}

void gauss(ll n, ll p) { //n x n+1-Matrix, Koerper F_p
    for (ll line = 0; line < n; line++) {
        normalLine(n, line, p);
        takeAll(n, line, p);
    }
}
```

4.4 Binomialkoeffizienten

```
ll calc_binom(ll N, ll K) {
   ll r = 1, d;
   if (K > N) return 0;
   for (d = 1; d <= K; d++) {
      r *= N--;
      r /= d;
   }
   return r;
}
```

4.5 Satz von Sprague-Grundy

Weise jedem Zustand X wie folgt eine Grundy-Zahl $g(X)$ zu:

$$g(X) := \min\{\mathbb{Z}_0^+ \setminus \{g(Y) \mid Y \text{ von } X \text{ aus direkt erreichbar}\}\}$$

X ist genau dann gewonnen, wenn $g(X) > 0$ ist.

Wenn man k Spiele in den Zuständen $X_1, \ldots, X_k$ hat, dann ist die Grundy-Zahl des Gesamtzustandes $g(X_1) \oplus \ldots \oplus g(X_k)$.

4.6 Maximales Teilfeld

```
//N :=  length of field
int maxStart = 1, maxLen = 0, curStart = 1, len = 0;
double maxValue = 0, sum = 0;
for (int pos = 0; pos < N; pos++) {
    sum += values[pos];
    len++;
    if (sum > maxValue) { // neues Maximum
        maxValue = sum; maxStart = curStart; maxLen = len;
    }
    if (sum < 0) { // alles zuruecksetzen
        curStart = pos +2; len = 0; sum = 0;
    }
}
//maxSum := maximaler Wert, maxStart := Startposition, maxLen := Laenge der Sequenz
```

Obiger Code findet kein maximales Teilfeld, das über das Ende hinausgeht. Dazu:
1. finde maximales Teilfeld, das nicht übers Ende geht
2. berechne minimales Teilfeld, das nicht über den Rand geht (analog)
3. nimm Maximum aus gefundenem Maximalem und Allem\Minimalem

5 Strings

5.1 Knuth-Morris-Pratt-Algorithmus

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

using namespace std;

//Preprocessing Substring sub for KMP-Search
vector<int> kmp_preprocessing(string& sub) {
    vector<int> b(sub.size() + 1);
    b[0] = -1;
    int i = 0, j = -1;
    while(i < sub.size()) {
        while(j >= 0 && sub[i] != sub[j])
            j = b[j];
        i++; j++;
        b[i] = j;
    }
    return b;
}

//Searching after Substring sub in s
vector<int> kmp_search(string& s, string& sub) {
    vector<int> pre = kmp_preprocessing(sub);
    vector<int> result;
    int i = 0, j = -1;
    while(i < s.size()) {
        while(j >= 0 && s[i] != sub[j])
            j = pre[j];
        i++; j++;
        if(j == sub.size()) {
            result.push_back(i-j);
            j = pre[j];
        }
    }
    return result;
}
```

5.2 Trie

```
//nur fuer kleinbuchstaben!
struct node {
    node *(e)[26];
    int c = 0;//anzahl der woerter die an dem node enden.
    node() { for(int i = 0; i < 26; i++) e[i] = NULL; }
};

void insert(node *root, string *txt, int s) {
    if(s >= txt->length()) root->c++;
    else {
        int idx = (int)((*txt).at(s) - 'a');
        if(root->e[idx] == NULL) {
            root->e[idx] = new node();
        }
        insert(root->e[idx], txt, s+1);
    }
}

int contains(node *root, string *txt, int s) {
    if(s >= txt->length()) return root->c;
    int idx = (int)((*txt).at(s) - 'a');
    if(root->e[idx] != NULL) {
            return contains(root->e[idx], txt, s+1);
    } else return 0;
}
```

5.3 Suffix-Array

```
//longest common substring in one string (overlapping not excluded)
//contains suffix array:--------------------------------------------------------------------
int cmp(string &s,vector<vector<int>> &v, int i, int vi, int u, int l) {
    int vi2 = (vi + 1) % 2, u2 = u + i / 2, l2 = l + i / 2;
    if(i == 1) return s[u] - s[l];
    else if (v[vi2][u] != v[vi2][l]) return (v[vi2][u] - v[vi2][l]);
    else { //beide groesser tifft nicht mehr ein, da ansonsten vorher schon unterschied in Laenge
        if(u2 >= s.length()) return -1;
        else if(l2 >= s.length()) return 1;
        else return v[vi2][u2] - v[vi2][l2];
    }
}

string lcsub(string s) {
    if(s.length() == 0) return "";
    vector<int> a(s.length());
    vector<vector<int>> v(2, vector<int>(s.length(), 0));
    int vi = 0;
    for(int k = 0; k < a.size(); k++) a[k] = k;
    for(int i = 1; i <= s.length(); i *= 2, vi = (vi + 1) % 2) {
        sort(a.begin(), a.end(), [&] (const int &u, const int &l) {
            return cmp(s, v, i, vi, u, l) < 0;
        });
        v[vi][a[0]] = 0;
        for(int z = 1; z < a.size(); z++) v[vi][a[z]] = v[vi][a[z-1]] + (cmp(s, v, i, vi, a[z], a[z-1]) == 0 ?
              0 : 1);
    }
//---------------------------------------------------------------------------------------------
    int r = 0, m=0, c=0;
    for(int i = 0; i < a.size() - 1; i++) {
        c = 0;
        while(a[i]+c < s.length() && a[i+1]+c < s.length() && s[a[i]+c] == s[a[i+1]+c]) c++;
        if(c > m) r=i, m=c;
    }
    return m == 0 ? "" : s.substr(a[r], m);
}
```

5.4 Longest Common Substring

```
//longest common substring.
struct lcse {
    int i = 0, s = 0;
};
string lcp(string s[2]) {
    if(s[0].length() == 0 || s[1].length() == 0) return "";
    vector<lcse> a(s[0].length()+s[1].length());
    for(int k = 0; k < a.size(); k++) a[k].i=(k < s[0].length() ? k : k - s[0].length()), a[k].s = (k < s[0].
         length() ? 0 : 1);
    sort(a.begin(), a.end(), [&] (const lcse &u, const lcse &l) {
        int ui = u.i, li = l.i;
        while(ui < s[u.s].length() && li < s[l.s].length()) {
            if(s[u.s][ui] < s[l.s][li]) return true;
            else if(s[u.s][ui] > s[l.s][li]) return false;
            ui++; li++;
        }
        return !(ui < s[u.s].length());
    });
    int r = 0, m=0, c=0;
    for(int i = 0; i < a.size() - 1; i++) {
        if(a[i].s == a[i+1].s) continue;
        c = 0;
        while(a[i].i+c < s[a[i].s].length() && a[i+1].i+c < s[a[i+1].s].length() && s[a[i].s][a[i].i+c] == s[a
             [i+1].s][a[i+1].i+c]) c++;
        if(c > m) r=i, m=c;
    }
    return m == 0 ? "" : s[a[r].s].substr(a[r].i, m);
}
```

5.5 Longest Common Subsequence

```
string lcss(string &a, string &b) {
    int m[a.length() + 1][b.length() + 1], x=0, y=0;
    memset(m, 0, sizeof(m));
    for(int y = a.length() - 1; y >= 0; y--) {
        for(int x = b.length() - 1; x >= 0; x--) {
            if(a[y] == b[x]) m[y][x] = 1 + m[y+1][x+1];
            else m[y][x] = max(m[y+1][x], m[y][x+1]);
        }
    } //for length only: return m[0][0];
    string res;
    while(x < b.length() && y < a.length()) {
        if(a[y] == b[x]) res += a[y++], x++;
        else if(m[y][x+1] > m[y+1][x+1]) x++;
        else y++;
    }
    return res;
}
```

6 Java

6.1 Introduction

- Compilen: `javac main.java`
- Ausführen: `java main < sample.in`
- Einlesen:

```
Scanner in = new Scanner(System.in); //java.util.Scanner
String line = in.nextLine(); //reads the next line of the input
int num = in.nextInt(); //reads the next token of the input as an int
double num2 = in.nextDouble(); //reads the next token of the input as a double
```

6.2 BigInteger

Hier ein kleiner überblick über die Methoden der Klasse BigInteger:

```
//Returns this +,*,/,- val
BigInteger add(BigInteger val), multiply(BigInteger val), divide(BigInteger val), substract(BigInteger val)

//Returns this\(^e\)
BigInteger pow(BigInteger e)

//Bit-Operations
BigInteger and(BigInteger val), or(BigInteger val), xor(BigInteger val), not(), shiftLeft(int n), shiftRight(
    int n)

//Returns the greatest common divisor of abs(this) and abs(val)
BigInteger gcd(BigInteger val)

//Returns this mod m, this\(^{-1}\) mod m, this\(^e\) mod m
BigInteger mod(BigInteger m), modInverse(BigInteger m), modPow(BigInteger e, BigInteger m)

//Returns the next number that is greater than this and that is probably a prime.
BigInteger  nextProbablePrime()

//Converting BigInteger. Attention: If the BigInteger is to big the lowest bits were choosen which fits into
    the converted data-type.
int intValue(), long longValue(), float floatValue(), double doubleValue()
```

7 Sonstiges

7.1 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.

7.2 Linear time sorting

```
int p[10] = {1,10,100,1000,10000,100000,1000000,10000000,100000000,1000000000};

int get_i_th_digit(int digit, int i) {
    return (digit / p[i]) % 10;
}

void sortLinear(vector<int> &s) {
    int max_digit;
    for(int i = 0; i < s.size(); i++) {
        int digit = ceil(log10(s[i]));
        if(digit > max_digit)
            max_digit = digit;
    }
    for(int d = 0; d < max_digit; d++) {
        vector<vector<int>> bucket(10);
        for(int i = 0; i < s.size(); i++)
            bucket[get_i_th_digit(s[i],d)].push_back(s[i]);
        s.clear();
        for(int i = 0; i < 10; i++)
            copy(bucket[i].begin(), bucket[i].end(), back_inserter(s));
    }
}
```