Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

22. November 2014

Inhaltsverzeichnis

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]);
}
```

2 Graphen

2.1 Kürzeste Wege

2.1.1 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()) {
  di front = pq.top(); pq.pop();
  int curNode = front.second, curDist = front.first;

  if (curDist > dist[curNode]) continue;

  for (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));
    }
  }
}
```

2.1.2 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 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
}
```

2.3 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);
  }
}
```

2.4 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;
}
```

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 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;
}
```

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;
}
```

4.1.2 Faktorisierung

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

using namespace std;

typedef unsigned long long ll;

const ll PRIME_SIZE = 10000000;
vector<int> primes;

//Call before calculating anything
void primeSieve() {
  vector<int> isPrime(PRIME_SIZE,true);
  for(ll i = 2; i < PRIME_SIZE; i+=2) {
    if(isPrime[i]) {
      primes.push_back(i);
      if(i*i <= PRIME_SIZE) {
        for(ll j = i; i*j < PRIME_SIZE; j+=2) isPrime[i*j] = false;
      }
    }
    if(i == 2)
      i--;
  }
}

//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;

}
```

4.1.3 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;
}
```

4.2 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);
  }
}
```

4.3 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;
}
```

4.4 Primzahlsieb von Eratosthenes

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

using namespace std;

typedef unsigned long long ll;

vector<int> primeSieve(ll n) {
  vector<int> primes;
  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--;
  }
  return primes;
}
```

5 Sonstiges

5.1 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.