

Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

22. November 2014

Inhaltsverzeichnis

1	Datenstrukturen	2
1.1	Union-Find	2
1.2	Segmentbaum	2
2	Graphen	2
2.1	Kürzeste Wege	2
2.1.1	Algorithmus von DIJKSTRA	2
2.1.2	BELLMANN-FORD-Algorithmus	3
2.2	Strongly Connected Components (TARJANS-Algorithmus)	3
2.3	Max-Flow (EDMONDS-KARP-Algorithmus)	4
3	Geometrie	4
3.1	Closest Pair	4
3.2	Geraden	5
3.3	Formeln - <code>std::complex</code>	6
4	Mathe	7
4.1	ggT, kgV, erweiterter euklidischer Algorithmus	7
4.1.1	Multiplikatives Inverses von x in $\mathbb{Z}/n\mathbb{Z}$	7
4.1.2	Primzahlsieb von Eratosthenes	7
4.1.3	Faktorisierung	7
4.1.4	Mod-Exponent über \mathbb{F}_p	8
4.2	LGS über \mathbb{F}_p	8
4.3	Binomialkoeffizienten	9
5	Sonstiges	9
5.1	2-SAT	9

1 Datenstrukturen

1.1 Union-Find

```

1 vector<int> parent, rank2; //manche compiler verbieten Variable mit Namen rank
2
3 int findSet(int n) { //Pfadkompression
4     if (parent[n] != n) parent[n] = findSet(parent[n]);
5     return parent[n];
6 }
7
8 void linkSets(int a, int b) { //union by rank
9     if (rank2[a] < rank2[b]) parent[a] = b;
10    else if (rank2[b] < rank2[a]) parent[b] = a;
11    else {
12        parent[a] = b;
13        rank2[b]++;
14    }
15 }
16
17 void unionSets(int a, int b) {
18     if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
19 }

```

1.2 Segmentbaum

```

1 int a[MAX_N], m[4 * MAX_N];
2
3 int query(int x, int y, int k = 0, int X = 0, int Y = MAX_N - 1) {
4     if (x <= X && Y <= y) return m[k];
5     if (y < X || Y < x) return -1000000000; //ein "neutrales" Element
6     int M = (X + Y) / 2;
7     return max(query(x, y, 2 * k + 1, X, M), query(x, y, 2 * k + 2, M + 1, Y));
8 }
9
10 void update(int i, int v, int k = 0, int X = 0, int Y = MAX_N - 1) {
11     if (i < X || Y < i) return;
12     if (X == Y) {
13         m[k] = v;
14         a[i] = v;
15         return;
16     }
17     int M = (X + Y) / 2;
18     update(i, v, 2 * k + 1, X, M);
19     update(i, v, 2 * k + 2, M + 1, Y);
20     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
21 }
22
23 void init(int k = 0, int X = 0, int Y = MAX_N - 1) {
24     if (X == Y) {
25         m[k] = a[X];
26         return;
27     }
28     int M = (X + Y) / 2;
29     init(2 * k + 1, X, M);
30     init(2 * k + 2, M + 1, Y);
31     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
32 }

```

2 Graphen

2.1 Kürzeste Wege

2.1.1 Algorithmus von DIJKSTRA

Kürzeste Pfade in Graphen ohne negative Kanten.

```

1 priority_queue<ii, vector<ii>, greater<ii> > pq;
2 vector<int> dist;
3 dist.assign(NUM_VERTICES, INF);

```

```

4 dist[0] = 0;
5 pq.push(ii(0, 0));
6
7 while (!pq.empty()) {
8     di front = pq.top(); pq.pop();
9     int curNode = front.second, curDist = front.first;
10
11     if (curDist > dist[curNode]) continue;
12
13     for (i = 0; i < (int)adjlist[curNode].size(); i++) {
14         int nextNode = adjlist[curNode][i].first, nextDist = curDist + adjlist[curNode][i].second;
15
16         if (nextDist < dist[nextNode]) {
17             dist[nextNode] = nextDist; pq.push(ii(nextDist, nextNode));
18         }
19     }
20 }

```

2.1.2 BELLMANN-FORD-Algorithmus

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

```

1 //n = number of vertices, edges is vector of edges
2 dist.assign(n, INF); dist[0] = 0;
3 parent.assign(n, -1);
4 for (i = 0; i < n - 1; i++) {
5     for (j = 0; j < (int)edges.size(); j++) {
6         if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
7             dist[edges[j].to] = dist[edges[j].from] + edges[j].cost;
8             parent[edges[j].to] = edges[j].from;
9         }
10     }
11 }
12 //now dist and parent are correct shortest paths
13 //next lines check for negative cycles
14 for (j = 0; j < (int)edges.size(); j++) {
15     if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
16         //NEGATIVE CYCLE found
17     }
18 }

```

2.2 Strongly Connected Components (TARJANS-Algorithmus)

```

1 int counter, sccCounter, n; //n == number of vertices
2 vector<bool> visited, inStack;
3 vector<vector<int>> adjlist;
4 vector<int> d, low, sccs;
5 stack<int> s;
6
7 void visit(int v) {
8     visited[v] = true;
9     d[v] = counter;
10    low[v] = counter;
11    counter++;
12    inStack[v] = true;
13    s.push(v);
14
15    for (int i = 0; i < (int)adjlist[v].size(); i++) {
16        int u = adjlist[v][i];
17        if (!visited[u]) {
18            visit(u);
19            low[v] = min(low[v], low[u]);
20        } else if (inStack[u]) {
21            low[v] = min(low[v], low[u]);
22        }
23    }
24
25    if (d[v] == low[v]) {
26        int u;

```

```

27     do {
28         u = s.top();
29         s.pop();
30         inStack[u] = false;
31         sccs[u] = sccCounter;
32     } while(u != v);
33     sccCounter++;
34 }
35 }
36
37 void scc() {
38     //read adjlist
39
40     visited.clear(); visited.assign(n, false);
41     d.clear(); d.resize(n);
42     low.clear(); low.resize(n);
43     inStack.clear(); inStack.assign(n, false);
44     sccs.clear(); sccs.resize(n);
45
46     counter = 0;
47     sccCounter = 0;
48     for (i = 0; i < n; i++) {
49         if (!visited[i]) {
50             visit(i);
51         }
52     }
53     //sccs has the component for each vertex
54 }

```

2.3 Max-Flow (EDMONDS-KARP-Algorithmus)

```

1  int s, t, f; //source, target, single flow
2  int res[MAX_V][MAX_V]; //adj-matrix
3  vector< vector<int> > adjList;
4  int p[MAX_V]; //bfs spanning tree
5
6  void augment(int v, int minEdge) {
7      if (v == s) { f = minEdge; return; }
8      else if (p[v] != -1) {
9          augment(p[v], min(minEdge, res[p[v]][v]));
10         res[p[v]][v] -= f; res[v][p[v]] += f;
11     }
12
13  int maxFlow() { //first initialize res, adjList, s and t
14      int mf = 0;
15      while (true) {
16          f = 0;
17          bitset<MAX_V> vis; vis[s] = true;
18          queue<int> q; q.push(s);
19          memset(p, -1, sizeof(p));
20          while (!q.empty()) { //BFS
21              int u = q.front(); q.pop();
22              if (u == t) break;
23              for (int j = 0; j < (int)adjList[u].size(); j++) {
24                  int v = adjList[u][j];
25                  if (res[u][v] > 0 && !vis[v]) {
26                      vis[v] = true; q.push(v); p[v] = u;
27                  }
28              }
29          }
30          augment(t, INF); //add found path to max flow
31          if (f == 0) break;
32          mf += f;
33      }
34      return mf;
35  }

```

3 Geometrie

3.1 Closest Pair

```

1 double squaredDist(point a, point b) {
2     return (a.first-b.first) * (a.first-b.first) + (a.second-b.second) * (a.second-b.second);
3 }
4
5 bool compY(point a, point b) {
6     if (a.second == b.second) return a.first < b.first;
7     return a.second < b.second;
8 }
9
10 double shortestDist(vector<point> &points) {
11     //check that points.size() > 1 and that ALL POINTS ARE DIFFERENT
12     set<point, bool(*)>(point, point)> status(compY);
13     sort(points.begin(), points.end());
14     double opt = 1e30, sqrtOpt = 1e15;
15     auto left = points.begin(), right = points.begin();
16     status.insert(*right); right++;
17
18     while (right != points.end()) {
19         if (fabs(left->first - right->first) >= sqrtOpt) {
20             status.erase(*(left++));
21         } else {
22             auto lower = status.lower_bound(point(-1e20, right->second - sqrtOpt));
23             auto upper = status.upper_bound(point(-1e20, right->second + sqrtOpt));
24             while (lower != upper) {
25                 double cand = squaredDist(*right, *lower);
26                 if (cand < opt) {
27                     opt = cand;
28                     sqrtOpt = sqrt(opt);
29                 }
30                 ++lower;
31             }
32             status.insert(*(right++));
33         }
34     }
35     return sqrtOpt;
36 }

```

3.2 Geraden

```

1 struct pt { //complex<double> does not work here, because we need to set pt.x and pt.y
2     double x, y;
3     pt() {};
4     pt(double x, double y) : x(x), y(y) {};
5 };
6
7 struct line {
8     double a, b, c; //a*x+b*y+c, b=0 ==> vertical line, b=1 ==> otherwise
9 };
10
11 line pointsToLine(pt p1, pt p2) {
12     line l;
13     if (fabs(p1.x - p2.x) < EPSILON) {
14         l.a = 1; l.b = 0.0; l.c = -p1.x;
15     } else {
16         l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
17         l.b = 1.0;
18         l.c = -(double)(l.a * p1.x) - p1.y;
19     }
20     return l;
21 }
22
23 bool areParallel(line l1, line l2) {
24     return (fabs(l1.a - l2.a) < EPSILON) && (fabs(l1.b - l2.b) < EPSILON);
25 }
26
27 bool areSame(line l1, line l2) {
28     return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPSILON);
29 }
30
31 bool areIntersect(line l1, line l2, pt &p) {
32     if (areParallel(l1, l2)) return false;

```

```

33 p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
34 if (fabs(l1.b) > EPSILON) p.y = -(l1.a * p.x + l1.c);
35 else p.y = -(l2.a * p.x + l2.c);
36 return true;
37 }

```

3.3 Formeln - std::complex

```

1 //komplexe Zahlen als Darstellung fuer Punkte
2 typedef pt complex<double>;
3 //Winkel zwischen Punkt und x-Achse in [0, 2 * PI), Winkel zwischen a und b
4 double angle = arg (a), angle_a_b = arg (a - b);
5 //Punkt rotiert um Winkel theta
6 pt a_rotated = a * exp (pt (0, theta));
7 //Mittelpunkt des Dreiecks abc
8 pt centroid = (a + b + c) / 3;
9 //Skalarprodukt
10 double dot(pt a, pt b) {
11     return real(conj(a) * b);
12 }
13 //Kreuzprodukt, 0, falls kollinear
14 double cross(pt a, pt b) {
15     return imag(conj(a) * b);
16 }
17 //wenn Eckpunkte bekannt
18 double areaOfTriangle(pt a, pt b, pt c) {
19     return abs(cross(b - a, c - a)) / 2.0;
20 }
21 //wenn Seitenlaengen bekannt
22 double areaOfTriangle(double a, double b, double c) {
23     double s = (a + b + c) / 2;
24     return sqrt(s * (s-a) * (s-b) * (s-c));
25 }
26 // Sind die Dreiecke a1, b1, c1, and a2, b2, c2 aehnlich?
27 // Erste Zeile testet Aehnlichkeit mit gleicher Orientierung,
28 // zweite Zeile testst Aehnlichkeit mit unterschiedlicher Orientierung
29 bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
30     return (
31         (b2 - a2) * (c1 - a1) == (b1 - a1) * (c2 - a2) ||
32         (b2 - a2) * (conj (c1) - conj (a1)) == (conj (b1) - conj (a1)) * (c2 - a2)
33     );
34 }
35 //Linksknick von a->b nach a->c
36 double ccw(pt a, pt b, pt c) {
37     return cross(b - a, c - a); //<0 => falls Rechtsknick, 0 => kollinear, >0 => Linksknick
38 }
39 //Streckenschnitt, Strecken a-b und c-d
40 bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
41     if (ccw(a, b, c) == 0 && ccw(a, b, d) == 0) { //kollinear
42         double dist = abs(a - b);
43         return (abs(a - c) <= dist && abs(b - c) <= dist) || (abs(a - d) <= dist && abs(b - d) <= dist);
44     }
45     return ccw(a, b, c) * ccw(a, b, d) <= 0 && ccw(c, d, a) * ccw(c, d, b) <= 0;
46 }
47 //Entfernung von p zu a-b
48 double distToLine(pt a, pt b, pt p) {
49     return abs(cross(p - a, b - a)) / abs(b - a);
50 }
51 //liegt p auf a-b
52 bool pointOnLine(pt a, pt b, pt p) {
53     return abs(distToLine(a, b, p)) < EPSILON;
54 }
55 //testet, ob d in der gleichen Ebene liegt wie a, b, und c
56 bool isCoplanar(pt a, pt b, pt c, pt d) {
57     return (b - a) * (c - a) * (d - a) == 0;
58 }

```

4 Mathe

4.1 ggT, kgV, erweiterter euklidischer Algorithmus

```

1 ll gcd(ll a, ll b) {
2     return b == 0 ? a : gcd(b, a % b);
3 }
4
5 ll lcm(ll a, ll b) {
6     return a * (b / gcd(a, b)); //Klammern gegen Overflow
7 }

1 //Accepted in Aufgabe mit Forderung: |X|+|Y| minimal (primaer) und X<=Y (sekundaer)
2 //hab aber keinen Beweis dafuer :)
3 ll x, y, d; //a * x + b * y = d = ggT(a,b)
4 void extendedEuclid(ll a, ll b) {
5     if (!b) {
6         x = 1; y = 0; d = a; return;
7     }
8     extendedEuclid(b, a % b);
9     ll x1 = y; ll y1 = x - (a / b) * y;
10    x = x1; y = y1;
11 }

```

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 $\alpha x + \beta n \equiv \alpha x \equiv 1 \pmod{n}$
- $x^{-1} := \alpha \pmod{n}$

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

```

1 ll multInv(ll n, ll p) { //berechnet das multiplikative Inverse von n in F_p
2     extendedEuclid(n, p); //implementierung von oben
3     x += ((x / p) + 1) * p;
4     return x % p;
5 }

```

4.1.2 Primzahlsieb von Eratosthenes

```

1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 typedef unsigned long long ll;
7
8 vector<int> primeSieve(ll n) {
9     vector<int> primes;
10    vector<int> isPrime(n, true);
11    for(ll i = 2; i < n; i+=2) {
12        if(isPrime[i]) {
13            primes.push_back(i);
14            if(i*i <= n) {
15                for(ll j = i; i*j < n; j+=2) isPrime[i*j] = false;
16            }
17        }
18        if(i == 2)
19            i--;
20    }
21    return primes;
22 }

```

4.1.3 Faktorisierung

```

1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  typedef unsigned long long ll;
7
8  const ll PRIME_SIZE = 10000000;
9  vector<int> primes;
10
11 //Call before calculating anything
12 void primeSieve() {
13     vector<int> isPrime(PRIME_SIZE, true);
14     for(ll i = 2; i < PRIME_SIZE; i+=2) {
15         if(isPrime[i]) {
16             primes.push_back(i);
17             if(i*i <= PRIME_SIZE) {
18                 for(ll j = i; i*j < PRIME_SIZE; j+=2) isPrime[i*j] = false;
19             }
20         }
21         if(i == 2)
22             i++;
23     }
24 }
25
26 //Factorize the number n
27 vector<int> factorize(ll n) {
28     vector<int> factor;
29     ll num = n;
30     int pos = 0;
31     while(num != 1) {
32         if(num % primes[pos] == 0) {
33             num /= primes[pos];
34             factor.push_back(primes[pos]);
35         }
36         else
37             pos++;
38         if(primes[pos]*primes[pos] > n)
39             break;
40     }
41     if(num != 1)
42         factor.push_back(num);
43     return factor;
44 }
45 }

```

4.1.4 Mod-Exponent über \mathbb{F}_p

```

1 ll modPow(ll b, ll e, ll p) {
2     if (e == 0) return 1;
3     if (e == 1) return b;
4     ll half = modPow(b, e / 2, p), res = (half * half) % p;
5     if (e & 1) res *= b; res %= p;
6     return res;
7 }

```

4.2 LGS über \mathbb{F}_p

```

1 void normalLine(ll n, ll line, ll p) { //normalisiert Zeile line
2     ll factor = multInv(mat[line][line], p); //Implementierung von oben
3     for (ll i = 0; i <= n; i++) {
4         mat[line][i] *= factor;
5         mat[line][i] %= p;
6     }
7 }
8
9 void takeAll(ll n, ll line, ll p) { //zieht Vielfaches von line von allen anderen Zeilen ab
10    for (ll i = 0; i < n; i++) {
11        if (i == line) continue;
12        ll diff = mat[i][line]; //abziehen
13        for (ll j = 0; j <= n; j++) {

```



```
14     mat[i][j] -= (diff * mat[line][j]) % p;
15     while (mat[i][j] < 0) {
16         mat[i][j] += p;
17     }
18 }
19 }
20 }
21
22 void gauss(ll n, ll p) { //n x n+1-Matrix, Koerper F_p
23     for (ll line = 0; line < n; line++) {
24         normalLine(n, line, p);
25         takeAll(n, line, p);
26     }
27 }
```

4.3 Binomialkoeffizienten

```
1 ll calc_binom(ll N, ll K) {
2     ll r = 1, d;
3     if (K > N) return 0;
4     for (d = 1; d <= K; d++) {
5         r *= N--;
6         r /= d;
7     }
8     return r;
9 }
```

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.