Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

22. November 2014

Inhaltsverzeichnis

1 Datenstrukturen

1.1 Union-Find

```cpp
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

```cpp
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.

```cpp
priority_queue<ii, vector<ii>, greater<ii> > pq;
vector<int> dist;
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   Artikulationspunkte und Brücken

```
 1 vector< vector<int> > adjlist;
 2 vector<int> low;
 3 vector<int> d;
 4 vector<bool> isArtPoint;
 5 vector< vector<int> > bridges; //nur fuer Bruecken
 6 int counter = 0;
 7
 8 void visit(int v, int parent) {
 9   d[v] = low[v] = ++counter;
10   int numVisits = 0, maxlow = 0;
11
12   for (vector<int>::iterator vit = adjlist[v].begin(); vit != adjlist[v].end(); vit++) {
13     if (d[*vit] == 0) {
14       numVisits++;
15       visit(*vit, v);
16       if (low[*vit] > maxlow) {
17         maxlow = low[*vit];
18       }
19
20       if (low[*vit] > d[v]) { //nur fuer Bruecken
21         bridges[v].push_back(*vit); bridges[*vit].push_back(v);
22       }
23
24       low[v] = min(low[v], low[*vit]);
25     } else {
26       if (d[*vit] < low[v]) {
27         low[v] = d[*vit];
28       }
29     }
30   }
31
32   if (parent == -1) {
33     if (numVisits > 1) isArtPoint[v] = true;
34   } else {
35     if (maxlow >= d[v]) isArtPoint[v] = true;
36   }
37 }
38
39 void findArticulationPoints() {
40   low.clear(); low.resize(adjlist.size());
```

```
41    d.clear(); d.assign(adjlist.size(), 0);
42    isArtPoint.clear(); isArtPoint.assign(adjlist.size(), false);
43    bridges.clear(); isBridge.resize(adjlist.size()); //nur fuer Bruecken
44    for (int v = 0; v < (int)adjlist.size(); v++) {
45      if (d[v] == 0) visit(v, -1);
46    }
47 }
```

2.4 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.

```
 1 vector< vector<int> > adjlist;
 2 vector< vector<int> > otherIdx;
 3 vector<int> cycle;
 4 vector<int> validIdx;
 5
 6 void swapEdges(int n, int a, int b) { // Vertauscht Kanten mit Indizes a und b von Knoten n.
 7   int neighA = adjlist[n][a];
 8   int neighB = adjlist[n][b];
 9   int idxNeighA = otherIdx[n][a];
10   int idxNeighB = otherIdx[n][b];
11   swap(adjlist[n][a], adjlist[n][b]);
12   swap(otherIdx[n][a], otherIdx[n][b]);
13   otherIdx[neighA][idxNeighA] = b;
14   otherIdx[neighB][idxNeighB] = a;
15 }
16
17 void removeEdge(int n, int i) { // Entfernt Kante i von Knoten n (und die zugehoerige Rueckwaertskante)
      .
18   int other = adjlist[n][i];
19   if (other == n) { //Schlingen
20     validIdx[n]++;
21     return;
22   }
23   int otherIndex = otherIdx[n][i];
24   validIdx[n]++;
25   if (otherIndex != validIdx[other]) {
26     swapEdges(other, otherIndex, validIdx[other]);
27   }
28   validIdx[other]++;
29 }
30
31 //findet Eulerzyklus an Knoten n startend
32 //teste vorher, dass Graph zusammenhaengend ist! (isolierte Punkte sind ok)
33 //teste vorher, ob Eulerzyklus ueberhaupt existiert!
34 void euler(int n) {
35   while (validIdx[n] < (int)adjlist[n].size()) {
36     int nn = adjlist[n][validIdx[n]];
37     removeEdge(n, validIdx[n]);
38     euler(nn);
39   }
40   cycle.push_back(n); //Zyklus am Ende in cycle
41 }
```

2.5 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 inititalize 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     augment(t, INF); //add found path to max flow
30     if (f == 0) break;
31     mf += f;
32   }
33   return mf;
34 }
```

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, becuase 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   Konvexe Hülle

```
1 #include <algorithm>
2 #include <iostream>
3 #include <sstream>
4 #include <string>
5 #include <vector>
6 using namespace std;
7
8 struct point {
9   double x, y;
10   point(){} point(double x, double y) : x(x), y(y) {}
11   bool operator <(const point &p) const {
12     return x < p.x || (x == p.x && y < p.y);
13   }
14 };
15
16 // 2D cross product.
17 // Return a positive value, if OAB makes a counter-clockwise turn,
18 // negative for clockwise turn, and zero if the points are collinear.
19 double cross(const point &O, const point &A, const point &B){
20   double d = (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
21   if (fabs(d) < 1e-9) return 0.0;
22   return d;
23 }
24
25 // Returns a list of points on the convex hull in counter-clockwise order.
26 // Colinear points are not in the convex hull, if you want colinear points in the hull remove "=" in
        the CCW-Test
```

```cpp
27  // Note: the last point in the returned list is the same as the first one.
28  vector<point> convexHull(vector<point> P){
29    int n = P.size(), k = 0;
30    vector<point> H(2*n);
31
32    // Sort points lexicographically
33    sort(P.begin(), P.end());
34
35    // Build lower hull
36    for (int i = 0; i < n; i++) {
37      while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0.0) k--;
38      H[k++] = P[i];
39    }
40
41    // Build upper hull
42    for (int i = n-2, t = k+1; i >= 0; i--) {
43      while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0.0) k--;
44      H[k++] = P[i];
45    }
46
47    H.resize(k);
48    return H;
49  }
```

3.4   Formeln - `std::complex`

```cpp
 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 }
59 //berechnet den Flaecheninhalt eines Polygons (nicht selbstschneidend)
60 double areaOfPolygon(vector<pt> &polygon) { //jeder Eckpunkt nur einmal im Vektor
61   double res = 0; int n = polygon.size();
62   for (int i = 0; i < (int)polygon.size(); i++)
63     res += real(polygon[i]) * imag(polygon[(i + 1) % n]) - real(polygon[(i + 1) % n]) * imag(polygon[i
          ]);
64   return 0.5 * abs(res);
65 }
66 //testet, ob sich zwei Rechtecke (p1, p2) und (p3, p4) schneiden (jeweils gegenueberliegende Ecken)
67 bool rectIntersection(pt p1, pt p2, pt p3, pt p4) {
68   double minx12 = min(real(p1), real(p2)), maxx12 = max(real(p1), real(p2));
69   double minx34 = min(real(p3), real(p4)), maxx34 = max(real(p3), real(p4));
70   double miny12 = min(imag(p1), imag(p2)), maxy12 = max(imag(p1), imag(p2));
71   double miny34 = min(imag(p3), imag(p4)), maxy34 = max(imag(p3), imag(p4));
72   return (maxx12 >= minx34) && (maxx34 >= minx12) && (maxy12 >= miny34) && (maxy34 >= miny12);
73 }
74 //testet, ob ein Punkt im Polygon liegt (beliebige Polygone)
75 bool pointInPolygon(pt p, vector<pt> &polygon) { //jeder Eckpunkt nur einmal im Vektor
76   pt rayEnd = p + pt(1, 1000000);
77   int counter = 0, n = polygon.size();
78   for (int i = 0; i < n; i++) {
79     pt start = polygon[i], end = polygon[(i + 1) % n];
80     if (lineSegmentIntersection(p, rayEnd, start, end)) counter++;
81   }
82   return counter & 1;
83 }
```

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

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   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 }
```

4.1.3   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 $\mathbb{F}_p$

```cpp
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

```cpp
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

```cpp
#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 Strings

5.1 KNUTH-MORRIS-PRATT-Algorithmus

```
 1  #include <iostream>
 2  #include <vector>
 3
 4  using namespace std;
 5
 6  //Preprocessing Substring sub for KMP-Search
 7  vector<int> kmp_preprocessing(string& sub) {
 8    vector<int> b(sub.size() + 1);
 9    b[0] = -1;
10    int i = 0, j = -1;
11    while(i < sub.size()) {
12      while(j >= 0 && sub[i] != sub[j])
13        j = b[j];
14      i++; j++;
15      b[i] = j;
16    }
17    return b;
18  }
19
20  //Searching after Substring sub in s
21  vector<int> kmp_search(string& s, string& sub) {
22    vector<int> pre = kmp_preprocessing(sub);
23    vector<int> result;
24    int i = 0, j = -1;
25    while(i < s.size()) {
26      while(j >= 0 && s[i] != sub[j])
27        j = pre[j];
28      i++; j++;
29      if(j == sub.size()) {
30        result.push_back(i-j);
31        j = pre[j];
32      }
33    }
34    return result;
35  }
```

5.2   Trie

```
 1  //nur fuer kleinbuchstaben!
 2  struct node {
 3    node *(e)[26];
 4    int c = 0;//anzahl der woerter die an dem node enden.
 5    node() { for(int i = 0; i < 26; i++) e[i] = NULL; }
 6  };
 7
 8  void insert(node *root, string *txt, int s) {
 9    if(s >= txt->length()) root->c++;
10    else {
11      int idx = (int)((*txt).at(s) - 'a');
12      if(root->e[idx] == NULL) {
13        root->e[idx] = new node();
14      }
15      insert(root->e[idx], txt, s+1);
16    }
17  }
18
19  int contains(node *root, string *txt, int s) {
20    if(s >= txt->length()) return root->c;
21    int idx = (int)((*txt).at(s) - 'a');
22    if(root->e[idx] != NULL) {
23        return contains(root->e[idx], txt, s+1);
24    } else return 0;
25  }
```

5.3   Suffix-Array

```
 1  //longest common substring in one string (overlapping not excluded)
 2  //contains suffix array:-----------------------------------------------------------------
 3  int cmp(string &s,vector<vector<int>> &v, int i, int vi, int u, int l) {
 4    int vi2 = (vi + 1) % 2, u2 = u + i / 2, l2 = l + i / 2;
 5    if(i == 1) return s[u] - s[l];
```

```
6    else if (v[vi2][u] != v[vi2][l]) return (v[vi2][u] - v[vi2][l]);
7    else { //beide groesser tifft nicht mehr ein, da ansonsten vorher schon unterschied in Laenge
8      if(u2 >= s.length()) return -1;
9      else if(l2 >= s.length()) return 1;
10     else return v[vi2][u2] - v[vi2][l2];
11   }
12 }
13
14 string lcsub(string s) {
15   if(s.length() == 0) return "";
16   vector<int> a(s.length());
17   vector<vector<int>> v(2, vector<int>(s.length(), 0));
18   int vi = 0;
19   for(int k = 0; k < a.size(); k++) a[k] = k;
20   for(int i = 1; i <= s.length(); i *= 2, vi = (vi + 1) % 2) {
21     sort(a.begin(), a.end(), [&] (const int &u, const int &l) {
22       return cmp(s, v, i, vi, u, l) < 0;
23     });
24     v[vi][a[0]] = 0;
25     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);
26   }
27 //-------------------------------------------------------------------------------
28   int r = 0, m=0, c=0;
29   for(int i = 0; i < a.size() - 1; i++) {
30     c = 0;
31     while(a[i]+c < s.length() && a[i+1]+c < s.length() && s[a[i]+c] == s[a[i+1]+c]) c++;
32     if(c > m) r=i, m=c;
33   }
34   return m == 0 ? "" : s.substr(a[r], m);
35 }
```

5.4 Longest Common Substring

```
1  //longest common substring.
2  struct lcse {
3    int i = 0, s = 0;
4  };
5  string lcp(string s[2]) {
6    if(s[0].length() == 0 || s[1].length() == 0) return "";
7    vector<lcse> a(s[0].length()+s[1].length());
8    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);
9    sort(a.begin(), a.end(), [&] (const lcse &u, const lcse &l) {
10     int ui = u.i, li = l.i;
11     while(ui < s[u.s].length() && li < s[l.s].length()) {
12       if(s[u.s][ui] < s[l.s][li]) return true;
13       else if(s[u.s][ui] > s[l.s][li]) return false;
14       ui++; li++;
15     }
16     return !(ui < s[u.s].length());
17   });
18   int r = 0, m=0, c=0;
19   for(int i = 0; i < a.size() - 1; i++) {
20     if(a[i].s == a[i+1].s) continue;
21     c = 0;
22     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++;
23     if(c > m) r=i, m=c;
24   }
25   return m == 0 ? "" : s[a[r].s].substr(a[r].i, m);
26 }
```

5.5 Longest Common Subsequence

```
1  string lcss(string &a, string &b) {
2    int m[a.length() + 1][b.length() + 1], x=0, y=0;
3    memset(m, 0, sizeof(m));
4    for(int y = a.length() - 1; y >= 0; y--) {
5      for(int x = b.length() - 1; x >= 0; x--) {
6        if(a[y] == b[x]) m[y][x] = 1 + m[y+1][x+1];
```

```
 7          else m[y][x] = max(m[y+1][x], m[y][x+1]);
 8      }
 9    } //for length only: return m[0][0];
10    string res;
11    while(x < b.length() && y < a.length()) {
12      if(a[y] == b[x]) res += a[y++], x++;
13      else if(m[y][x+1] > m[y+1][x+1]) x++;
14      else y++;
15    }
16    return res;
17 }
```

6 Sonstiges

6.1 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.