Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

15. Oktober 2016

Inhaltsverzeichnis

1 Datenstrukturen

1.1 Union-Find

```
// Laufzeit: O(n*alpha(n))
// "height" ist obere Schranke für die Höhe der Bäume. Sobald
// Pfadkompression angewendet wurde, ist die genaue Höhe nicht mehr
// effizient berechenbar.
vector<int> parent // Initialisiere mit Index im Array.
vector<int> height; // Initialisiere mit 0.

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 (height[a] < height[b]) parent[a] = b;
  else if (height[b] < height[a]) parent[b] = a;
  else {
    parent[a] = b;
    height[b]++;
}}

void unionSets(int a, int b) { // Diese Funktion aufrufen.
  if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
}
```

1.2 Segmentbaum

```
// Laufzeit: init: O(n), query: O(log n), update: O(log n)
// Berechnet das Maximum im Array.
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 -INF; // 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]);
}
```

Mit `update()` können ganze Intervalle geändert werden. Dazu: Offset in den inneren Knoten des Baums speichern.

1.3 Fenwick Tree

```
vector<int> FT; // Fenwick-Tree
int n;

// Addiert val zum Element an Index i. O(log(n)).
void updateFT(int i, int val) {
  i++; while(i <= n) { FT[i] += val; i += (i & (-i)); }
}

// Baut Baum auf. O(n*log(n)).
void buildFenwickTree(vector<int>& a) {
  n = a.size();
  FT.assign(n+1,0);
  for(int i = 0; i < n; i++)  updateFT(i,a[i]);
}

// Präfix-Summe über das Intervall [0..i]. O(log(n)).
int prefix_sum(int i) {
  int sum = 0; i++;
  while(i > 0) { sum += FT[i]; i -= (i & (-i)); }
  return sum;
}
```

1.4 Range Minimum Query

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

// Baut Struktur auf. O(n*log(n))
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];
}}}}

// Gibt den Index des Minimums im Intervall [l,r) zurück. O(1).
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]);
}
```

1.5 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 = min i, mit X[i] >= 10
  return 0;
}
```

2 Graphen

2.1 Minimale Spannbäume

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. ($\Rightarrow$ 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.

2.1.1 Kruskal

```
// Union-Find Implementierung von oben. Laufzeit: O(|E|*log(|E|))
sort(edges.begin(), edges.end());
vector<ii> mst; int cost = 0;
for (auto &e : edges) {
  if (findSet(e.from) != findSet(e.to)) {
    unionSets(e.from, e.to);
    mst.push_back(ii(e.from, e.to));
    cost += e.cost;
}}
```

2.2 Kürzeste Wege

2.2.1 Algorithmus von Dijkstra

Kürzeste Pfade in Graphen ohne negative Kanten.

```
// Laufzeit: O((|E|+|V|)*log |V|)
void dijkstra(int start) {
  priority_queue<ii, vector<ii>, greater<ii> > pq;
  vector<int> dist(NUM_VERTICES, INF), parent(NUM_VERTICES, -1);
  dist[start] = 0; pq.push(ii(0, start));

  while (!pq.empty()) {
    ii front = pq.top(); pq.pop();
    int curNode = front.second, curDist = front.first;
    if (curDist > dist[curNode]) continue; // WICHTIG!

    for (auto n : adjlist[curNode]) {
      int nextNode = n.first, nextDist = curDist + n.second;
      if (nextDist < dist[nextNode]) {
        dist[nextNode] = nextDist; parent[nextNode] = curNode;
        pq.push(ii(nextDist, nextNode));
}}}}
```

2.2.2 Bellmann-Ford-Algorithmus

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

```
// Laufzeit: O(|V|*|E|)
vector<edge> edges; // Kanten einfügen!
vector<int> dist, parent;

void bellmannFord() {
  dist.assign(NUM_VERTICES, INF); dist[0] = 0;
  parent.assign(NUM_VERTICES, -1);
  for (int i = 0; i < NUM_VERTICES - 1; i++) {
    for (auto &e : edges) {
      if (dist[e.from] + e.cost < dist[e.to]) {
        dist[e.to] = dist[e.from] + e.cost;
        parent[e.to] = e.from;
  }}}

  // "dist" und "parent" sind korrekte kürzeste Pfade.
  // Folgende Zeilen prüfen nur negative Kreise.
  for (auto &e : edges) {
    if (dist[e.from] + e.cost < dist[e.to]) {
      // Negativer Kreis gefunden.
}}}
```

2.2.3 Floyd-Warshall-Algorithmus

```
// Initialisiere mat: mat[i][i] = 0, mat[i][j] = INF falls i & j nicht
// adjazent, Länge sonst. Laufzeit: O(|V|^3)
void floydWarshall() {
  for (k = 0; k < MAX_V; k++) {
    for (i = 0; i < MAX_V; i++) {
      for (j = 0; j < MAX_V; j++) {
        if (mat[i][k] != INF && mat[k][j] != INF && mat[i][k] + mat[k][j]
            < mat[i][j]) {
          mat[i][j] = mat[i][k] + mat[k][j];
}}}}}
```

- Nur negative Werte sollten die Nullen überschreiben.
- Von parallelen Kanten sollte nur die günstigste gespeichert werden.
- `i` liegt genau dann auf einem negativen Kreis, wenn `dist[i][i] < 0` ist.
- Wenn für `c` gilt, dass `dist[u][c] != INF && dist[c][v] != INF && dist[c][c] < 0`, wird der u-v-Pfad beliebig kurz.

2.3 Strongly Connected Components (Tarjans-Algorithmus)

```
// Laufzeit: O(|V|+|E|)
int counter, sccCounter;
vector<bool> visited, inStack;
vector< vector<int> > adjlist;
vector<int> d, low, sccs; // sccs enthält den Index der SCC pro Knoten.
stack<int> s;

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

  for (auto u : adjlist[v]) {
    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() {
  visited.assign(adjlist.size(), false);
  d.assign(adjlist.size(), -1);
  low.assign(adjlist.size(), -1);
  inStack.assign(adjlist.size(), false);
  sccs.resize(adjlist.size(), -1);

  counter = sccCounter = 0;
  for (int i = 0; i < (int)adjlist.size(); i++) {
    if (!visited[i]) {
      visit(i);
}}}
```

2.4 Artikulationspunkte und Brücken

```
// Laufzeit: O(|V|+|E|)
vector< vector<int> > adjlist;
vector<bool> isArt;
vector<int> d, low;
int counter, root; // root >= 2 <=> Wurzel Artikulationspunkt
vector<ii> bridges; // Nur fuer Brücken.

void dfs(int v, int parent) { // Mit parent=-1 aufrufen.
  d[v] = low[v] = counter++;
  if (parent == 0) root++;

  for (auto w : adjlist[v]) {
    if (!d[w]) {
      dfs(w, v);
      if (low[w] >= d[v]) isArt[v] = true;
      if (low[w] > d[v]) bridges.push_back(ii(v, w));
      low[v] = min(low[v], low[w]);
    } else if (w != parent) {
      low[v] = min(low[v], d[w]);
}}}

void findArticulationPoints() {
  couter = 1; // Nicht auf 0 setzen!
  low.resize(adjlist.size());
  d.assign(adjlist.size(), 0);
  isArtPoint.assign(adjlist.size(), false);
  bridges.clear(); //nur fuer Bruecken
  for (int v = 0; v < (int)adjlist.size(); v++) if (!d[v]) 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.
- Algorithmus schlägt nicht fehl, falls kein Eulerzyklus existiert. Die Existenz muss separat geprüft werden.

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

```
// Laufzeit: O(|V|+|E|)
vector< vector<int> > adjlist, otherIdx;
vector<int> cycle, validIdx;

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

// Entfernt Kante i von Knoten n (und die zugehörige Rückwärtskante).
void removeEdge(int n, int i) {
  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 zusammenhängend ist! Isolierten Knoten?
// Teste vorher, ob Eulerzyklus überhaupt 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 in cycle in umgekehrter Reihenfolge.
}
```

2.6 Lowest Common Ancestor

```
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) { // Laufzeit: O(n)
  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++;
}}

int getLCA(int a, int b) { // Laufzeit: O(1)
  return visited[queryRMQ(
      min(first[a], first[b]), max(first[a], first[b]))];
}

// Benutzung:
int c = 0;
initLCA(0, 0, c);
initRMQ(); // Ersetze das data im RMQ-Code von oben durch depth.
```

2.7 Max-Flow

2.7.1 Capacity Scaling

Gut bei dünn besetzten Graphen.

```
// Ford Fulkerson mit Capacity Scaling. Laufzeit: O(|E|^2*log(C))
struct MaxFlow { // Muss mit new erstellt werden!
  static const int MAX_N = 500; // #Knoten, egal für die Laufzeit.
  struct edge { int dest, rev; ll cap, flow; };
  vector<edge> adjlist[MAX_N];
  int visited[MAX_N] = {0}, target, dfsCounter = 0;
  ll capacity;

  bool dfs(int x) {
    if (x == target) return 1;
    if (visited[x] == dfsCounter) return 0;
    visited[x] = dfsCounter;
    for (edge &e : adjlist[x]) {
      if (e.cap >= capacity && dfs(e.dest)) {
        e.cap -= capacity; adjlist[e.dest][e.rev].cap += capacity;
```

```
        e.flow += capacity; adjlist[e.dest][e.rev].flow -= capacity;
        return 1;
    }}
    return 0;
  }

  void addEdge(int u, int v, ll c) {
    adjlist[u].push_back(edge {v, (int)adjlist[v].size(), c, 0});
    adjlist[v].push_back(edge {u, (int)adjlist[u].size() - 1, 0, 0});
  }

  ll maxFlow(int s, int t) {
    capacity = 1L << 62;
    target = t;
    ll flow = 0L;
    while (capacity) {
      while (dfsCounter++, dfs(s)) flow += capacity;
      capacity /= 2;
    }
    return flow;
  }
};
```

2.7.2 Push Relabel

Gut bei sehr dicht besetzten Graphen.

```
// Laufzeit: O(|V|^3)
struct PushRelabel {
  ll capacities[MAX_V][MAX_V], flow[MAX_V][MAX_V], excess[MAX_V];
  int height[MAX_V], list[MAX_V - 2], seen[MAX_V], n;

  PushRelabel(int n) {
    this->n = n;
    memset(capacities, 0L, sizeof(capacities));
    memset(flow, 0L, sizeof(flow));
    memset(excess, 0L, sizeof(excess));
    memset(height, 0, sizeof(height));
    memset(list, 0, sizeof(list));
    memset(seen, 0, sizeof(seen));
  }

  inline void addEdge(int u, int v, ll c) { capacities[u][v] += c; }

  void push(int u, int v) {
    ll send = min(excess[u], capacities[u][v] - flow[u][v]);
    flow[u][v] += send; flow[v][u] -= send;
    excess[u] -= send; excess[v] += send;
  }

  void relabel(int u) {
    int minHeight = INT_MAX / 2;
    for (int v = 0; v < n; v++) {
      if (capacities[u][v] - flow[u][v] > 0) {
        minHeight = min(minHeight, height[v]);
        height[u] = minHeight + 1;
  }}}

  void discharge(int u) {
    while (excess[u] > 0) {
      if (seen[u] < n) {
        int v = seen[u];
        if (capacities[u][v] - flow[u][v] > 0 && height[u] > height[v]) {
          push(u, v);
        } else seen[u]++;
      } else {
        relabel(u);
        seen[u] = 0;
  }}}

  void moveToFront(int u) {
    int temp = list[u];
    for (int i = u; i > 0; i--) list[i] = list[i - 1];
    list[0] = temp;
  }

  ll maxFlow(int source, int target) {
    for (int i = 0, p = 0; i < n; i++)
      if (i != source && i != target) list[p++] = i;

    height[source] = n;
    excess[source] = LLONG_MAX / 2;
    for (int i = 0; i < n; i++) push(source, i);

    int p = 0;
    while (p < n - 2) {
      int u = list[p], oldHeight = height[u];
      discharge(u);
      if (height[u] > oldHeight) {
        moveToFront(p);
        p = 0;
      } else p++;
    }

    ll maxflow = 0L;
    for (int i = 0; i < n; i++) maxflow += flow[source][i];
    return maxflow;
  }
};
```

2.7.3 Anwendungen

- **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 den unterschiedlichen Pfaden ohne gemeinsame Kanten.

- **Maximum Independent Paths**
  Finde die maximale Anzahl an Pfaden 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 den unterschiedlichen Pfaden ohne gemeinsame Knoten.
- **Min-Cut**
  Der maximale Fluss ist gleich dem minimalen Schnitt. Bei Quelle s und Senke t, partitioniere in S und T. Zu S gehören alle Knoten, die im Residualgraphen von s aus erreichbar sind (Rückwärtskanten beachten).

2.8 Min-Cost-Max-Flow

```
static const ll flowlimit = 1LL << 60; // Größer als der maximale Fluss.
struct MinCostFlow { // Mit new erstellen!
  static const int maxn = 400; // Größer als die Anzahl der Knoten.
  static const int maxm = 5000; // Größer als die Anzahhl der Kanten.
  struct edge { int node, next; ll flow, value; } edges[maxm << 1];
  int graph[maxn], queue[maxn], pre[maxn], con[maxn];
  int n, m, source, target, top;
  bool inqueue[maxn];
  ll maxflow, mincost, dis[maxn];

  MinCostFlow() { memset(graph, -1, sizeof(graph)); top = 0; }

  inline int inverse(int x) { return 1 + ((x >> 1) << 2) - x; }

  // Directed edge from u to v, capacity c, weight w.
  inline int addedge(int u, int v, int c, int w) {
    edges[top].value = w; edges[top].flow = c; edges[top].node = v;
    edges[top].next = graph[u]; graph[u] = top++;
    edges[top].value = -w; edges[top].flow = 0; edges[top].node = u;
    edges[top].next = graph[v]; graph[v] = top++;
    return top - 2;
  }

  bool SPFA() {
    int point, node, now, head = 0, tail = 1;
    memset(pre, -1, sizeof(pre));
    memset(inqueue, 0, sizeof(inqueue));
    memset(dis, 0x7F, sizeof(dis));
    dis[source] = 0; queue[0] = source;
    pre[source] = source; inqueue[source] = true;

    while (head != tail) {
      now = queue[head++];
      point = graph[now];
      inqueue[now] = false;
      head %= maxn;

      while (point != -1) {
        node = edges[point].node;
        if (edges[point].flow > 0 &&
            dis[node] > dis[now] + edges[point].value) {
          dis[node] = dis[now] + edges[point].value;
          pre[node] = now; con[node] = point;
          if (!inqueue[node]) {
            inqueue[node] = true; queue[tail++] = node;
            tail %= maxn;
        }}
        point = edges[point].next;
    }}
    return pre[target] != -1;
  }

  void extend() {
    ll w = flowlimit;
    for (int u = target; pre[u] != u; u = pre[u])
      w = min(w, edges[con[u]].flow);
    maxflow += w;
    mincost += dis[target] * w;
    for (int u = target; pre[u] != u; u = pre[u]) {
      edges[con[u]].flow -= w;
      edges[inverse(con[u])].flow += w;
  }}

  void mincostflow() {
    maxflow = mincost = 0;
    while (SPFA()) extend();
  }
};
```

2.9 Maximal Cardinatlity Bipartite Matching

```
// Laufzeit: O(n*min(ans^2, |E|))
vector< vector<int> > adjlist; // Von links nach rechts.
vector<int> pairs; // Der gematchte Knoten oder -1.
vector<bool> visited;

bool dfs(int v) {
  if (visited[v]) return false;
  visited[v] = true;
  for (auto w : adjlist[v]) if (pairs[w] < 0 || dfs(pairs[w])) {
    pairs[w] = v; pairs[v] = w; return true;
  }
  return false;
}

int kuhn(int n) { // n = #Knoten links.
  pairs.assign(adjlist.size(), -1);
  int ans = 0;
  // Greedy Matching. Optionale Beschleunigung.
  for (int i = 0; i < n; i++) for (auto w : adjlist[i])
    if (pairs[w] == -1) pairs[i] = w; pairs[w] = i; ans++; break; }
```

```
  for (int i = 0; i < n; i++) if (pairs[i] == -1) {
    visited.assign(n, false);
    ans += dfs(i);
  }
  return ans; // Größe des Matchings.
}
```

3 Geometrie

3.1 Closest Pair

```
double squaredDist(pt a, pt b) {
  return (a.fst-b.fst) * (a.fst-b.fst) + (a.snd-b.snd) * (a.snd-b.snd);
}

bool compY(pt a, pt b) {
  if (a.snd == b.snd) return a.fst < b.fst;
  return a.snd < b.snd;
}

// points.size() > 1 und alle Punkte müssen verschieden sein!
double shortestDist(vector<pt> &points) {
  set<pt, bool(*)(pt, pt)> 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->fst - right->fst) >= sqrtOpt) {
      status.erase(*(left++));
    } else {
      auto lower = status.lower_bound(pt(-1e20, right->snd - sqrtOpt));
      auto upper = status.upper_bound(pt(-1e20, right->snd + 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

```
// Nicht complex<double> benutzen. Eigene struct schreiben.
struct line {
  double a, b, c; // ax + by + c = 0; vertikale Line: b = 0, sonst: b = 1
};

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

```
// Laufzeit: O(n*log(n))

ll cross(const pt p, const pt a, const pt b) {
  return (a.x - p.x) * (b.y - p.y) - (a.y - p.y) * (b.x - p.x);
}

// Punkte auf der konvexen Hülle, gegen den Uhrzeigersinn sortiert.
// Kollineare Punkte nicht enthalten, entferne dafür "=" im CCW-Test.
// Achtung: Der erste und letzte Punkt im Ergebnis sind gleich.
// Achtung: Alle Punkte müssen verschieden sein.
vector<pt> convexHull(vector<pt> p){
  int n = p.size(), k = 0;
  vector<pt> h(2 * n);
  sort(p.begin(), p.end());
  for (int i = 0; i < n; i++) { // Untere Hülle.
    while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
    h[k++] = p[i];
```

```
  }
  for (int i = n - 2, t = k + 1; i >= 0; i--) { // Obere Hülle.
    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 für Punkte. Wenn immer möglich
// complex<int> verwenden. Funktionen wie abs() geben dann int zurück.
typedef pt complex<double>;

// Winkel zwischen Punkt und x-Achse in [0, 2 * PI), bzw. zwischen a, 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) / 3.0;

// 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); }

// Flächeninhalt eines Dreicks bei bekannten Eckpunkten.
double areaOfTriangle(pt a, pt b, pt c) {
  return abs(cross(b - a, c - a)) / 2.0;
}

// Flächeninhalt eines Dreiecks bei bekannten Seitenlängen.
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 ähnlich?
// Erste Zeile testet Ähnlichkeit mit gleicher Orientierung,
// zweite Zeile testet Ähnlichkeit 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)
  );
}

// -1 => gegen den Uhrzeigersinn, 0 => kolliniear, 1 => im Uhrzeigersinn.
// Einschränken der Rückgabe auf [-1,1] ist sicherer gegen Overflows.
double orientation(pt a, pt b, pt c) {
  double orien = cross(b - a, c - a);
  if (abs(orien) < EPSILON) return 0; // Braucht großes EPSILON: ~1e-6
  return orien < 0 ? -1 : 1;
}

// Test auf Streckenschnitt zwischen a-b und c-d.
bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
  if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0) {
    double dist = abs(a - b);
    return (abs(a - c) <= dist && abs(b - c) <= dist) ||
           (abs(a - d) <= dist && abs(b - d) <= dist);
  }
  return orientation(a, b, c) * orientation(a, b, d) <= 0 &&
         orientation(c, d, a) * orientation(c, d, b) <= 0;
}

// Berechnet die Schnittpunkte der Strecken a-b und c-d. Enthält entweder
// keinen Punkt, den einzigen Schnittpunkt oder die Endpunkte der
// Schnittstrecke. operator<, min, max müssen noch geschrieben werden!
vector<pt> lineSegmentIntersection(pt a, pt b, pt c, pt d) {
  vector<pt> result;
  if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0 &&
      orientation(c, d, a) == 0 && orientation(c, d, b) == 0) {
    pt minAB = min(a, b), maxAB = max(a, b);
    pt minCD = min(c, d), maxCD = max(c, d);
    if (minAB < minCD && maxAB < minCD) return result;
    if (minCD < minAB && maxCD < minAB) return result;
    pt start = max(minAB, minCD), end = min(maxAB, maxCD);
    result.push_back(start);
    if (start != end) result.push_back(end);
    return result;
  }
  double x1 = real(b) - real(a), y1 = imag(b) - imag(a);
  double x2 = real(d) - real(c), y2 = imag(d) - imag(c);
  double u1 = (-y1 * (real(a) - real(c)) + x1 * (imag(a) - imag(c))) /
      (-x2 * y1 + x1 * y2);
  double u2 = (x2 * (imag(a) - imag(c)) - y2 * (real(a) - real(c))) /
      (-x2 * y1 + x1 * y2);
  if (u1 >= 0 && u1 <= 1 && u2 >= 0 && u2 <= 1) {
    double x = real(a) + u2 * x1, y = imag(a) + u2 * y1;
    result.push_back(pt(x, y));
  }
  return result;
}

// Entfernung von Punkt p zur Gearden durch a-b.
double distToLine(pt a, pt b, pt p) {
  return abs(cross(p - a, b - a)) / abs(b - a);
}

// Liegt p auf der Geraden a-b?
bool pointOnLine(pt a, pt b, pt p) {
  return orientation(a, b, c) == 0;
}
```

```
98
99  // Liegt p auf der Strecke a-b?
100 bool pointOnLineSegment(pt a, pt b, pt p) {
101   if (orientation(a, b, p) != 0) return false;
102   return real(p) >= min(real(a), real(b)) &&
103          real(p) <= max(real(a), real(b)) &&
104          imag(p) >= min(imag(a), imag(b)) &&
105          imag(p) <= max(imag(a), imag(b));
106 }
107
108 // Entfernung von Punkt p zur Strecke a-b.
109 double distToSegment(pt a, pt b, pt p) {
110   if (a == b) return abs(p - a);
111     double segLength = abs(a - b);
112     double u = ((real(p) - real(a)) * (real(b) - real(a)) +
113         (imag(p) - imag(a)) * (imag(b) - imag(a))) /
114         (segLength * segLength);
115     pt projection(real(a) + u * (real(b) - real(a)),
116         imag(a) + u * (imag(b) - imag(a)));
117     double projectionDist = abs(p - projection);
118     if (!pointOnLineSegment(a, b, projection)) projectionDist = 1e30;
119     return min(projectionDist, min(abs(p - a), abs(p - b)));
120 }
121
122 // Kürzeste Entfernung zwischen den Strecken a-b und c-d.
123 double distBetweenSegments(pt a, pt b, pt c, pt d) {
124   if (lineSegmentIntersection(a, b, c, d)) return 0.0;
125   double result = distToSegment(a, b, c);
126   result = min(result, distToSegment(a, b, d));
127   result = min(result, distToSegment(c, d, a));
128   return min(result, distToSegment(c, d, b));
129 }
130
131 // Liegt d in der gleichen Ebene wie a, b, und c?
132 bool isCoplanar(pt a, pt b, pt c, pt d) {
133   return abs((b - a) * (c - a) * (d - a)) < EPSILON;
134 }
135
136 // Berechnet den Flächeninhalt eines Polygons (nicht selbstschneidend).
137 // Punkte gegen den Uhrzeigersinn: positiv, sonst negativ.
138 double areaOfPolygon(vector<pt> &polygon) { // Jeder Eckpunkt nur einmal.
139   double res = 0; int n = polygon.size();
140   for (int i = 0; i < n; i++)
141     res += real(polygon[i]) * imag(polygon[(i + 1) % n]) -
142            real(polygon[(i + 1) % n]) * imag(polygon[i]);
143   return 0.5 * res;
144 }
145
146 // Schneiden sich (p1, p2) und (p3, p4) (gegenüberliegende Ecken).
147 bool rectIntersection(pt p1, pt p2, pt p3, pt p4) {
148   double minx12=min(real(p1), real(p2)), maxx12=max(real(p1), real(p2));
149   double minx34=min(real(p3), real(p4)), maxx34=max(real(p3), real(p4));
150   double miny12=min(imag(p1), imag(p2)), maxy12=max(imag(p1), imag(p2));
151   double miny34=min(imag(p3), imag(p4)), maxy34=max(imag(p3), imag(p4));
152   return (maxx12 >= minx34) && (maxx34 >= minx12) &&
153          (maxy12 >= miny34) && (maxy34 >= miny12);
154 }
155
156 // Testet, ob ein Punkt im Polygon liegt (beliebige Polygone).
157 bool pointInPolygon(pt p, vector<pt> &polygon) { // Punkte nur einmal.
158   pt rayEnd = p + pt(1, 1000000);
159   int counter = 0, n = polygon.size();
160   for (int i = 0; i < n; i++) {
161     pt start = polygon[i], end = polygon[(i + 1) % n];
162     if (lineSegmentIntersection(p, rayEnd, start, end)) counter++;
163   }
164   return counter & 1;
165 }
```

4 Mathe

4.1 ggT, kgV, erweiterter euklidischer Algorithmus

```
1 // Laufzeiten: O(log(a) + log(b))
2 ll gcd(ll a, ll b) { return b == 0 ? a : gcd (b, a % b); }
3 ll lcm(ll a, ll b) { return a * (b / gcd(a, b)); }
```

```
1 ll extendedEuclid(ll a, ll b, ll &x, ll &y) { // a*x + b*y = ggt(a, b).
2   if (a == 0) { x = 0; y = 1; return b; }
3   ll x1, y1, d = extendedEuclid(b % a, a, x1, y1);
4   x = y1 - (b / a) * x1; y = x1;
5   return d;
6 }
```

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

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

```
1 // Laufzeit: O(log (n) + log(p))
2 ll multInv(ll n, ll p) {
3   ll x, y;
4   extendedEuclid(n, p, x, y); // Implementierung von oben.
5   x += ((x / p) + 1) * p;
6   return x % p;
7 }
```

4.2 Mod-Exponent über $\mathbb{F}_p$

```cpp
// Laufzeit: O(log(b))
ll multMod(ll a, ll b, ll n) {
  if(a == 0 || b == 0) return 0;
  if(b == 1) return a % n;
  if(b % 2 == 1) return (a + multMod(a, b-1, n)) % n;
  else return multMod((a + a) % n, b / 2, n);
}

// Laufzeit: O(log(b))
ll powMod(ll a, ll b, ll n) {
  if(b == 0) return 1;
  if(b == 1) return a % n;
  if(b % 2 == 1) return multMod(powMod(a, b-1, n), a, n);
  else return powMod(multMod(a, a, n), b / 2, n);
}
```

4.3 LGS über $\mathbb{F}_p$

```cpp
// Laufzeit: O(n^3)
void swapLines(int n, int l1, int l2) {
  for (int i = 0; i <= n; i++) swap(mat[l1][i], mat[l2][i]);
}

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

void takeAll(int n, int line, ll p) {
  for (int i = 0; i < n; i++) {
    if (i == line) continue;
    ll diff = mat[i][line];
    for (int j = 0; j <= n; j++) {
      mat[i][j] -= (diff * mat[line][j]) % p;
      mat[i][j] %= p;
      if (mat[i][j] < 0) mat[i][j] += p;
}}}

void gauss(int n, ll p) { // nx(n+1)-Matrix, Körper F_p.
  for (int line = 0; line < n; line++) {
    int swappee = line;
    while (mat[swappee][line] == 0) swappee++;
    swapLines(n, line, swappee);
    normalLine(n, line, p);
    takeAll(n, line, p);
}}
```

4.4 LGS über $\mathbb{R}$

```cpp
// Laufzeit: O(n^3)
void swapLines(int n, int l1, int l2) {
  for (int i = 0; i <= n; i++) swap(mat[l1][i], mat[l2][i]);
}

void normalLine(int n, int line) {
  double factor = mat[line][line];
  for (int i = 0; i <= n; i++) {
    mat[line][i] /= factor;
}}

void takeAll(int n, int line) {
  for (int i = 0; i < n; i++) {
    if (i == line) continue;
    double diff = mat[i][line];
    for (int j = 0; j <= n; j++) {
      mat[i][j] -= diff * mat[line][j];
}}}

int gauss(int n) { // Gibt zurück, ob das System (eindeutig) lösbar ist.
  vector<bool> done(n, false);
  for (int i = 0; i < n; i++) {
    int swappee = i; // Sucht Pivotzeile für bessere Stabilität.
    for (int j = 0; j < n; j++) {
      if (done[j]) continue;
      if (abs(mat[j][i]) > abs(mat[i][i])) swappee = j;
    }
    swapLines(n, i, swappee);
    if (abs(mat[i][i]) > EPSILON) {
      normalLine(n, i);
      takeAll(n, i);
      done[i] = true;
  }} // Ab jetzt nur noch checks bzgl. Eindeutigkeit/Existenz der Lösung.
  for (int i = 0; i < n; i++) {
    bool allZero = true;
    for (int j = i; j < n; j++)
      if (abs(mat[i][j]) > EPSILON) allZero = false;
    if (allZero && abs(mat[i][n]) > EPSILON) return INCONSISTENT;
    if (allZero && abs(mat[i][n]) < EPSILON) return MULTIPLE;
  }
  return UNIQUE;
}
```

4.5 Chinesischer Restsatz

- Extrem anfällig gegen Overflows. Evtl. häufig 128-Bit Integer verwenden.
- Direkte Formel für zwei Kongruenzen $x \equiv a \mod n, x \equiv b \mod m$:

$$x \equiv a - y * n * \frac{a-b}{d} \mod \frac{mn}{d} \quad \text{mit} \quad d := ggT(n,m) = yn + zm$$

Formel kann auch für nicht teilerfremde Moduli verwendet werden. Sind

die Moduli nicht teilerfremd, existiert genau dann eine Lösung, wenn $a_i \equiv a_j \mod \gcd(m_i, m_j)$. In diesem Fall sind keine Faktoren auf der linken Seite erlaubt.

```
// Laufzeit: O(n * log(n)), n := Anzahl der Kongruenzen
// Nur für teilerfremde Moduli. Berechnet das kleinste, nicht negative x,
// das alle Kongruenzen simultan löst. Alle Lösungen sind kongruent zum
// kgV der Moduli (Produkt, falls alle teilerfremd sind).
struct ChineseRemainder {
  typedef __int128 lll;
  vector<lll> lhs, rhs, mods, inv;
  lll M; // Produkt über die Moduli. Kann leicht überlaufen.

  ll g(vector<lll> &vec) {
    lll res = 0;
    for (int i = 0; i < (int)vec.size(); i++) {
      res += (vec[i] * inv[i]) % M;
      res %= M;
    }
    return res;
  }

  // Fügt Kongruenz l * x = r (mod m) hinzu.
  void addEquation(ll l, ll r, ll m) {
    lhs.push_back(l);
    rhs.push_back(r);
    mods.push_back(m);
  }

  // Löst das System.
  ll solve() {
    M = accumulate(mods.begin(), mods.end(), lll(1), multiplies<lll>());
    inv.resize(lhs.size());
    for (int i = 0; i < (int)lhs.size(); i++) {
      lll x = (M / mods[i]) % mods[i];
      inv[i] = (multInv(x, mods[i]) * (M / mods[i]));
    }
    return (multInv(g(lhs), M) * g(rhs)) % M;
  }
};
```

4.6 Primzahlsieb von Eratosthenes

```
// Laufzeit: O(n * log log n)
#define N 100000001 // Bis 10^8 in unter 64MB Speicher.
bitset<N / 2> isPrime;

inline bool check(int x) { // Diese Methode zum Lookup verwenden.
  if (x < 2) return false;
  else if (x == 2) return true;
  else if (!(x & 1)) return false;
  else return !isPrime[x / 2];
}

inline int primeSieve(int n) { // Rückgabe: Anzahl der Primzahlen <= n.
  int counter = 1;
  for (int i = 3; i <= min(N, n); i += 2) {
    if (!isPrime[i / 2]) {
      for (int j = 3 * i; j <= min(N, n); j+= 2 * i) isPrime[j / 2] = 1;
      counter++;
  }}
  return counter;
}
```

4.7 Miller-Rabin-Primzahltest

```
// Laufzeit: O(log n). Exakt, nicht propabilistisch.
bool isPrime(ll n) {
  if(n == 2) return true;
  if(n < 2 || n % 2 == 0) return false;
  ll d = n - 1, j = 0;
  while(d % 2 == 0) d >>= 1, j++;
  for(int a = 2; a <= min((ll)37, n - 1); a++) {
    ll v = powMod(a, d, n); // Implementierung von oben.
    if(v == 1 || v == n - 1) continue;
    for(int i = 1;  i <= j; i++) {
      v = multMod(v, v, n); // Implementierung von oben.
      if(v == n - 1 || v <= 1) break;
    }
    if(v != n - 1) return false;
  }
  return true;
}
```

4.8 Binomialkoeffizienten

Vorberechnen, wenn häufig benötigt.

```
// Laufzeit: O(k)
ll calc_binom(ll n, ll k) { // Sehr sicher gegen Overflows.
   ll r = 1, d;
   if (k > n) return 0;
   for (d = 1; d <= k; d++) { // Reihenfolge garantiert Teilbarkeit.
      r *= n--;
      r /= d;
   }
   return r;
}
```

4.9 Polynome & FFT

Multipliziert Polynome A und B.

- $\deg(A * B) = \deg(A) + \deg(B)$
- Vektoren `a` und `b` müssen mindestens Größe $\deg(A * B) + 1$ haben. Größe muss eine Zweierpotenz sein.
- Für ganzzahlige Koeffizienten: `(int)round(real(a[i]))`

```
// Laufzeit: O(n log(n)).
typedef complex<double> cplx; // Eigene Implementierung ist schneller.

// a.size() muss eine Zweierpotenz sein!
vector<cplx> fft(const vector<cplx> &a, bool inverse = 0) {
  int logn = 1, n = a.size();
  vector<cplx> A(n);
  while ((1 << logn) < n) logn++;
  for (int i = 0; i < n; i++) {
    int j = 0;
    for (int k = 0; k < logn; k++) j = (j << 1) | ((i >> k) & 1);
    A[j] = a[i];
  }
  for (int s = 2; s <= n; s <<= 1) {
    double angle = 2 * PI / s * (inverse ? -1 : 1);
    cplx ws(cos(angle), sin(angle));
    for (int j = 0; j < n; j+= s) {
      cplx w = 1;
      for (int k = 0; k < s / 2; k++) {
        cplx u = A[j + k], t = A[j + s / 2 + k];
        A[j + k] = u + w * t;
        A[j + s / 2 + k] = u - w * t;
        if (inverse) A[j + k] /= 2, A[j + s / 2 + k] /= 2;
        w *= ws;
  }}}
  return A;
}

// Polynome: a[0] = a_0, a[1] = a_1, ... und b[0] = b_0, b[1] = b_1, ...
// Bei Integern: Runde Koeffizienten: (int)round(a[i].real())
vector<cplx> a = {0,0,0,0,1,2,3,4}, b = {0,0,0,0,2,3,0,1};
a = fft(a); b = fft(b);
for (int i = 0; i < (int)a.size(); i++) a[i] *= b[i];
a = fft(a,1); // a = a * b
```

4.10 3D-Kugeln

```
// Great Cirlce Distance mit Längen- und Breitengrad.
double gcDist(
    double pLat, double pLon, double qLat, double qLon, double radius) {
  pLat *= PI / 180; pLon *= PI / 180; qLat *= PI / 180; qLon *= PI / 180;
  return radius * acos(cos(pLat) * cos(pLon) * cos(qLat) * cos(qLon) +
                       cos(pLat) * sin(pLon) * cos(qLat) * sin(qLon) +
                       sin(pLat) * sin(qLat));
}

// Great Cirlce Distance mit kartesischen Koordinaten.
double gcDist(point p, point q) {
  return acos(p.x * q.x + p.y * q.y + p.z * q.z);
}

// 3D Punkt in kartesischen Koordinaten.
struct point{
  double x, y, z;
  point() {}
  point(double x, double y, double z) : x(x), y(y), z(z) {}
  point(double lat, double lon) {
    lat *= PI / 180.0; lon *= PI / 180.0;
    x = cos(lat) * sin(lon); y = cos(lat) * cos(lon); z = sin(lat);
  }
};
```

4.11 Kombinatorik

4.11.1 Berühmte Zahlen

Fibonacci-Zahlen	$f(0) = 0 \quad f(1) = 1 \quad f(n+2) = f(n+1) + f(n)$
Catalan-Zahlen	$C_0 = 1 \quad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1}\binom{2n}{n} = \frac{2(2n-1)}{n+1} \cdot C_{n-1}$
Euler-Zahlen (I)	$\left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1 \quad \left\langle {n \atop k} \right\rangle = (k+1)\left\langle {n-1 \atop k} \right\rangle + (n-k)\left\langle {n-1 \atop k-1} \right\rangle$
Euler-Zahlen (II)	$\left\langle\left\langle {n \atop 0} \right\rangle\right\rangle = 1 \quad \left\langle\left\langle {n \atop n} \right\rangle\right\rangle = 0 \quad \left\langle\left\langle {n \atop k} \right\rangle\right\rangle = (k+1)\left\langle\left\langle {n-1 \atop k} \right\rangle\right\rangle + (2n-k-1)\left\langle$
Stirling-Zahlen (I)	$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1 \quad \begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0 \quad \begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$
Stirling-Zahlen (II)	$\begin{Bmatrix} n \\ 1 \end{Bmatrix} = \begin{Bmatrix} n \\ n \end{Bmatrix} = 1 \quad \begin{Bmatrix} n \\ k \end{Bmatrix} = k\begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}$
Integer-Partitions	$f(1,1) = 1 \quad f(n,k) = 0 \text{ für } k > n \quad f(n,k) = f(n-k,k) + f(n,$

Bemerkung 1 $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix}$

Bemerkung 2 (Zeckendorfs Theorem) *Jede positive natürliche Zahl kann eindeutig als Summe einer oder mehrerer verschiedener* Fibonacci-*Zahlen geschrieben werden, sodass keine zwei aufeinanderfolgenden* Fibonacci-*Zahlen in der Summe vorkommen.*

Lösung: *Greedy, nimm immer die größte* Fibonacci-*Zahl, die noch hineinpasst.*

Bemerkung 3

- *Die erste und dritte angegebene Formel sind relativ sicher gegen Overflows.*
- *Die erste Formel kann auch zur Berechnung der* Catalan-*Zahlen bezüglich eines Moduls genutzt werden.*

Bemerkung 4 *Die* Catalan-*Zahlen geben an:* $C_n =$

- *Anzahl der Binärbäume mit n nicht unterscheidbaren Knoten.*
- *Anzahl der validen Klammerausdrücke mit n Klammerpaaren.*
- *Anzahl der korrekten Klammerungen von n + 1 Faktoren.*
- *Anzahl der Möglichkeiten ein konvexes Polygon mit n + 2 Ecken in Dreiecke zu zerlegen.*
- *Anzahl der monotonen Pfade (zwischen gegenüberliegenden Ecken) in einem n×n-Gitter, die nicht die Diagonale kreuzen.*

Bemerkung 5 (Euler-Zahlen 1. Ordnung) *Die Anzahl der Permutationen von* $\{1,\dots,n\}$ *mit genau k Anstiegen.*

Begründung: Für die n-te Zahl gibt es n mögliche Positionen zum Einfügen. Dabei wird entweder ein Ansteig in zwei gesplitted oder ein Anstieg um n ergänzt.

Bemerkung 6 (Euler-Zahlen 2. Ordnung) *Die Anzahl der Permutationen von* $\{1,1,\dots,n,n\}$ *mit genau k Anstiegen.*

Bemerkung 7 (Stirling-Zahlen 1. Ordnung) *Die Anzahl der Permutationen von* $\{1,\dots,n\}$ *mit genau k Zyklen.*

Begründung: Es gibt zwei Möglichkeiten für die n-te Zahl. Entweder sie bildet einen eigene Zyklus, oder sie kann an jeder Position in jedem Zyklus einsortiert werden.

Bemerkung 8 (Stirling-Zahlen 2. Ordnung) *Die Anzahl der Möglichkeiten n Elemente in k nichtleere Teilmengen zu zerlegen.*

Begründung: Es gibt k Möglichkeiten die n in eine n−1-Partition einzuordnen. Dazu kommt der Fall, dass die n in ihrer eigenen Teilmenge (alleine) steht.

Bemerkung 9 *Anzahl der Teilmengen von* $\mathbb{N}$*, die sich zu n aufaddieren mit maximalem Elment* $\leq k$*.*

4.11.2 Verschiedenes

Türme von Hanoi, minimale Schirttzahl:	$T_n = 2^n - 1$
#Regionen zwischen n Gearden	$n(n+1)/2 + 1$
#Abgeschlossene Regionen zwischen n Geraden	$\left(n^2 - 3n + 2\right)/2$
#Markierte, gewurzelte Bäume	n^{n-1}
#Markierte, nicht gewurzelte Bäume	n^{n-2}

4.12 Satz von Sprague-Grundy

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

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

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

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

4.13 Big Integers

```
// Bislang keine Division. Multiplikation nach Schulmethode.
#define PLUS    0
#define MINUS   1
#define BASE    1000000000
#define EXPONET 9

struct bigint {
  int sign;
  vector<ll> digits;

  // Initialisiert mit 0.
  bigint(void) { sign = PLUS; }

  // Initialisiert mit kleinem Wert.
  bigint(ll value) {
    if (value == 0) sign = PLUS;
    else {
      sign = value >= 0 ? PLUS : MINUS;
      value = abs(value);
      while (value) {
        digits.push_back(value % BASE);
        value /= BASE;
  }}}

  // Initialisiert mit C-String. Kann nicht mit Vorzeichen umgehen.
  bigint(char *str, int length) {
    int base = 1;
    ll digit = 0;
    for (int i = length - 1; i >= 0; i--) {
      digit += base * (str[i] - '0');
      if (base * 10 == BASE) {
        digits.push_back(digit);
        digit = 0;
        base = 1;
      } else base *= 10;
    }
    if (digit != 0) digits.push_back(digit);
    sign = PLUS;
  }

  // Löscht führende Nullen und macht -0 zu 0.
  void trim() {
    while (digits.size() > 0 && digits[digits.size() - 1] == 0)
        digits.pop_back();
    if (digits.size() == 0 && sign == MINUS) sign = PLUS;
  }
```

```
47
48    // Gibt die Zahl aus.
49    void print() {
50      if (digits.size() == 0) { printf("0"); return; }
51      if (sign == MINUS) printf("-");
52      printf("%lld", digits[digits.size() - 1]);
53      for (int i = digits.size() - 2; i >= 0; i--) {
54        printf("%09lld", digits[i]); // Anpassen, wenn andere Basis gewählt
                wird.
55    }}
56  };
57
58  // Kleiner-oder-gleich-Vergleich.
59  bool operator<=(bigint &a, bigint &b) {
60    if (a.digits.size() == b.digits.size()) {
61      int idx = a.digits.size() - 1;
62      while (idx >= 0) {
63        if (a.digits[idx] < b.digits[idx]) return true;
64        else if (a.digits[idx] > b.digits[idx]) return false;
65        idx--;
66      }
67      return true;
68    }
69    return a.digits.size() < b.digits.size();
70  }
71
72  // Kleiner-Vergeleich.
73  bool operator<(bigint &a, bigint &b) {
74    if (a.digits.size() == b.digits.size()) {
75      int idx = a.digits.size() - 1;
76      while (idx >= 0) {
77        if (a.digits[idx] < b.digits[idx]) return true;
78        else if (a.digits[idx] > b.digits[idx]) return false;
79        idx--;
80      }
81      return false;
82    }
83    return a.digits.size() < b.digits.size();
84  }
85
86  void sub(bigint *a, bigint *b, bigint *c);
87
88  // a + b = c. a, b, c dürfen gleich sein.
89  void add(bigint *a, bigint *b, bigint *c) {
90    if (a->sign == b->sign) c->sign = a->sign;
91    else {
92      if (a->sign == MINUS) {
93        a->sign ^= 1;
94        sub(b, a, c);
95        a->sign ^= 1;
96      } else {
97        b->sign ^= 1;
98        sub(a, b, c);
99        b->sign ^= 1;
100     }
101     return;
102   }
103
104   c->digits.resize(max(a->digits.size(), b->digits.size()));
105   ll carry = 0;
106   int i = 0;
107   for (; i < (int)min(a->digits.size(), b->digits.size()); i++) {
108     ll sum = carry + a->digits[i] + b->digits[i];
109     c->digits[i] = sum % BASE;
110     carry = sum / BASE;
111   }
112   if (i < (int)a->digits.size()) {
113     for (; i< (int)a->digits.size(); i++) {
114       ll sum = carry + a->digits[i];
115       c->digits[i] = sum % BASE;
116       carry = sum / BASE;
117     }
118   } else {
119     for (; i< (int)b->digits.size(); i++) {
120       ll sum = carry + b->digits[i];
121       c->digits[i] = sum % BASE;
122       carry = sum / BASE;
123   }}
124   if (carry) c->digits.push_back(carry);
125 }
126
127 // a - b = c. c darf a oder b sein. a und b müssen verschieden sein.
128 void sub(bigint *a, bigint *b, bigint *c) {
129   if (a->sign == MINUS || b->sign == MINUS) {
130     b->sign ^= 1;
131     add(a, b, c);
132     b->sign ^= 1;
133     return;
134   }
135
136   if (a < b) {
137     sub(b, a, c);
138     c->sign = MINUS;
139     c->trim();
140     return;
141   }
142
143   c->digits.resize(a->digits.size());
144   ll borrow = 0;
145   int i = 0;
146   for (; i < (int)b->digits.size(); i++) {
147     ll diff = a->digits[i] - borrow - b->digits[i];
148     if (a->digits[i] > 0) borrow = 0;
149     if (diff < 0) {
150       diff += BASE;
151       borrow = 1;
152     }
153     c->digits[i] = diff % BASE;
154   }
155   for (; i < (int)a->digits.size(); i++) {
```

```
    ll diff = a->digits[i] - borrow;
    if (a->digits[i] > 0) borrow = 0;
    if (diff < 0) {
      diff += BASE;
      borrow = 1;
    }
    c->digits[i] = diff % BASE;
  }
  c->trim();
}

// Zifferнmultiplikation a * b = c. b und c dürfen gleich sein.
// a muss kleiner BASE sein.
void digitMul(ll a, bigint *b, bigint *c) {
  if (a == 0) {
    c->digits.clear();
    c->sign = PLUS;
    return;
  }
  c->digits.resize(b->digits.size());
  ll carry = 0;
  for (int i = 0; i < (int)b->digits.size(); i++) {
    ll prod = carry + b->digits[i] * a;
    c->digits[i] = prod % BASE;
    carry = prod / BASE;
  }
  if (carry) c->digits.push_back(carry);
  c->sign = (a > 0) ? b->sign : 1 ^ b->sign;
  c->trim();
}

// Zifferndivision b / a = c. b und c dürfen gleich sein.
// a muss kleiner BASE sein.
void digitDiv(ll a, bigint *b, bigint *c) {
  c->digits.resize(b->digits.size());
  ll carry = 0;
  for (int i = (int)b->digits.size() - 1; i>= 0; i--) {
    ll quot = (carry * BASE + b->digits[i]) / a;
    carry = carry * BASE + b->digits[i] - quot * a;
    c->digits[i] = quot;
  }
  c->sign = b->sign ^ (a < 0);
  c->trim();
}

// a * b = c. c darf weder a noch b sein. a und b dürfen gleich sein.
void mult(bigint *a, bigint *b, bigint *c) {
  bigint row = *a, tmp;
  c->digits.clear();
  for (int i = 0; i < (int)b->digits.size(); i++) {
    digitMul(b->digits[i], &row, &tmp);
    add(&tmp, c, c);
    row.digits.insert(row.digits.begin(), 0);
  }
  c->sign = a->sign != b->sign;
  c->trim();
}

// Berechnet eine kleine Zehnerpotenz.
inline ll pow10(int n) {
  ll res = 1;
  for (int i = 0; i < n; i++) res *= 10;
  return res;
}

// Berechnet eine große Zehnerpotenz.
void power10(ll e, bigint *out) {
  out->digits.assign(e / EXPONET + 1, 0);
  if (e % EXPONET)
    out->digits[out->digits.size() - 1] = pow10(e % EXPONET);
  else out->digits[out->digits.size() - 1] = 1;
}

// Nimmt eine Zahl module einer Zehnerpotenz 10^e.
void mod10(int e, bigint *a) {
  int idx = e / EXPONET;
  if ((int)a->digits.size() < idx + 1) return;
  if (e % EXPONET) {
    a->digits.resize(idx + 1);
    a->digits[idx] %= pow10(e % EXPONET);
  } else {
    a->digits.resize(idx);
  }
  a->trim();
}
```

5 Strings

5.1 Knuth-Morris-Pratt-Algorithmus

```
// Laufzeit: O(n + m), n = #Text, m = #Pattern
vector<int> kmpPreprocessing(string &sub) {
  vector<int> b(sub.length() + 1);
  b[0] = -1;
  int i = 0, j = -1;
  while (i < (int)sub.length()) {
    while (j >= 0 && sub[i] != sub[j]) j = b[j];
    i++; j++;
    b[i] = j;
  }
  return b;
}

vector<int> kmpSearch(string &s, string &sub) {
  vector<int> pre = kmpPreprocessing(sub), result;
  int i = 0, j = 0;
  while (i < (int)s.length()) {
```

```
    while (j >= 0 && s[i] != sub[j]) j = pre[j];
    i++; j++;
    if (j == (int)sub.length()) {
      result.push_back(i - j);
      j = pre[j];
  }}
  return result;
}
```

5.2 Aho-Corasick-Automat

```
// Laufzeit: O(n + m + z), n = #Text, m = Summe #Pattern, z = #Matches
// Findet mehrere Patterns gleichzeitig in einem String.
// 1) Wurzel erstellen: vertex *automaton = new vertex();
// 2) Mit addString(automaton, s, idx); Patterns hinzufügen.
// 3) finishAutomaton(automaton) aufrufen.
// 4) Mit automaton = go(automaton, c) in nächsten Zustand wechseln.
//    DANACH: Wenn patterns-Vektor nicht leer ist: Hier enden alle
//    enthaltenen Patterns.
// ACHTUNG: Die Zahlenwerte der auftretenden Buchstaben müssen
// zusammenhängend sein und bei 0 beginnen!
struct vertex {
  vertex *next[ALPHABET_SIZE], *failure;
  char character;
  vector<int> patterns; // Indizes der Patterns, die hier enden.
  vertex() { for (int i = 0; i < ALPHABET_SIZE; i++) next[i] = NULL; }
};

void addString(vertex *v, string &pattern, int patternIdx) {
  for (int i = 0; i < (int)pattern.length(); i++) {
    if (!v->next[(int)pattern[i]]) {
      vertex *w = new vertex();
      w->character = pattern[i];
      v->next[(int)pattern[i]] = w;
    }
    v = v->next[(int)pattern[i]];
  }
  v->patterns.push_back(patternIdx);
}

void finishAutomaton(vertex *v) {
  for (int i = 0; i < ALPHABET_SIZE; i++)
    if (!v->next[i]) v->next[i] = v;

  queue<vertex*> q;
  for (int i = 0; i < ALPHABET_SIZE; i++) {
    if (v->next[i] != v) {
      v->next[i]->failure = v;
      q.push(v->next[i]);
  }}
  while (!q.empty()) {
    vertex *r = q.front(); q.pop();
    for (int i = 0; i < ALPHABET_SIZE; i++) {
      if (r->next[i]) {
        q.push(r->next[i]);
        vertex *f = r->failure;
        while (!f->next[i]) f = f->failure;
        r->next[i]->failure = f->next[i];
        for (int j = 0; j < (int)f->next[i]->patterns.size(); j++) {
          r->next[i]->patterns.push_back(f->next[i]->patterns[j]);
}}}}}

vertex* go(vertex *v, char c) {
  if (v->next[(int)c]) return v->next[(int)c];
  else return go(v->failure, c);
}
```

5.3 Trie

```
struct node {
  node *(e)[26]; // Implementierung für Kleinbuchstaben.
  int c = 0; // Anzahl der Wörter, die an diesem node enden.
  node() { for(int i = 0; i < 26; i++) e[i] = NULL; }
};

void insert(node *root, string &txt, int s) { // Laufzeit: O(|txt|)
  if(s == (int)txt.size()) root->c++;
  else {
    int idx = (int)(txt[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) { // Laufzeit: O(|txt|)
  if(s == txt.size()) return root->c;
  int idx = (int)(txt[s] - 'a');
  if(root->e[idx] != NULL) return contains(root->e[idx], txt, s + 1);
  else return 0;
}
```

5.4 Suffix-Array

```
struct SuffixArray { // MAX_LG = ceil(log2(MAX_N))
  static const int MAX_N = 100010, MAX_LG = 17;
  pair<pair<int, int>, int> L[MAX_N];
  int P[MAX_LG + 1][MAX_N], n, step, count;
  int suffixArray[MAX_N], lcpArray[MAX_N];

  SuffixArray(const string &s) : n(s.size()) { // Laufzeit: O(n*log^2(n))
    for (int i = 0; i < n; i++) P[0][i] = s[i];
    suffixArray[0] = 0; // Falls n == 1.
    for (step = 1, count = 1; count < n; step++, count <<= 1) {
      for (int i = 0; i < n; i++) L[i] =
```

```
          {{P[step-1][i], i+count < n ? P[step-1][i+count] : -1}, i};
      sort(L, L + n);
      for (int i = 0; i < n; i++) P[step][L[i].second] = i > 0 &&
          L[i].first == L[i-1].first ? P[step][L[i-1].second] : i;
    }
    for (int i = 0; i < n; i++) suffixArray[i] = L[i].second;
    for (int i = 1; i < n; i++)
      lcpArray[i] = lcp(suffixArray[i - 1], suffixArray[i]);
  }

  // x und y sind Indizes im String, nicht im Suffixarray.
  int lcp(int x, int y) { // Laufzeit: O(log(n))
    int k, ret = 0;
    if (x == y) return n - x;
    for (k = step - 1; k >= 0 && x < n && y < n; k--)
      if (P[k][x] == P[k][y])
        x += 1 << k, y += 1 << k, ret += 1 << k;
    return ret;
  }
};
```

5.5 Suffix-Automaton

```
#define ALPHABET_SIZE   26
struct SuffixAutomaton {
  struct State {
    int length; int link; int next[ALPHABET_SIZE];
    State() { memset(next, 0, sizeof(next)); }
  };
  static const int MAX_N = 100000; // Maximale Länge des Strings.
  State states[2 * MAX_N];
  int size, last;

  SuffixAutomaton(string &s) { // Laufzeit: O(|s|)
    size = 1; last = 0;
    states[0].length = 0;
    states[0].link = -1;
    for (auto c : s) extend(c);
  }

  void extend(char c) { // Werte von c müssen bei 0 beginnen.
    c -= 'a';
    int current = size++;
    states[current].length = states[last].length + 1;
    int pos = last;
    while (pos != -1 && !states[pos].next[(int)c]) {
      states[pos].next[(int)c] = current;
      pos = states[pos].link;
    }
    if (pos == -1) states[current].link = 0;
    else {
      int q = states[pos].next[(int)c];
      if (states[pos].length + 1 == states[q].length) {
        states[current].link = q;
      } else {
        int clone = size++;
        states[clone].length = states[pos].length + 1;
        states[clone].link = states[q].link;
        memcpy(states[clone].next, states[q].next,
            sizeof(states[q].next));
        while (pos != -1 && states[pos].next[(int)c] == q) {
          states[pos].next[(int)c] = clone;
          pos = states[pos].link;
        }
        states[q].link = states[current].link = clone;
    }}
    last = current;
  }

  // Paar mit Startposition und Länge des LCS. Index in Parameter s.
  ii longestCommonSubstring(string &s) { // Laufzeit: O(|s|)
    int v = 0, l = 0, best = 0, bestpos = 0;
    for (int i = 0; i < (int)s.size(); i++) {
      int c = s[i] - 'a';
      while (v && !states[v].next[c]) {
        v = states[v].link;
        l = states[v].length;
      }
      if (states[v].next[c]) { v = states[v].next[c]; l++; }
      if (l > best) { best = l; bestpos = i; }
    }
    return ii(bestpos - best + 1, best);
  }
};
```

5.6 Longest Common Subsequence

```
// Laufzeit: O(|a|*|b|)
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]);
  }} // Für die Länge: 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`
- Eingabe: `Scanner` ist sehr langsam. Bei großen Eingaben muss ein Buffered Reader verwendet werden.

```
Scanner in = new Scanner(System.in); // java.util.Scanner
String line = in.nextLine(); // Die nächste Zeile.
int num = in.nextInt(); // Das nächste Token als int.
double num2 = in.nextDouble(); // Das nächste Token als double.
```

- Ausgabe:

```
// In StringBuilder schreiben und auf einmal ausgeben ist schneller.
StringBuilder sb = new StringBuilder(); // java.lang.StringBuilder
sb.append("Hallo Welt");
System.out.print(sb.toString());
```

6.2 BigInteger

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

// Berechnet this^e.
BigInteger pow(BigInteger e)

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

// Berechnet den ggT von abs(this) und abs(val).
BigInteger gcd(BigInteger val)

// Berechnet this mod m, this^-1 mod m, this^e mod m.
BigInteger mod(BigInteger m), modInverse(BigInteger m),
           modPow(BigInteger e, BigInteger m)

// Berechnet die nächste Zahl, die größer und wahrscheinlich prim ist.
BigInteger  nextProbablePrime()

// Berechnet int/long/float/double-Wert.
// Ist die Zahl zu großen werden die niedrigsten Bits konvertiert.
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 Zeileneingabe

```
// Zerlegt s anhand aller Zeichen in delim.
vector<string> split(string &s, string delim) {
  vector<string> result; char *token;
  token = strtok((char*)s.c_str(), (char*)delim.c_str());
  while (token != NULL) {
    result.push_back(string(token));
    token = strtok(NULL, (char*)delim.c_str());
  }
  return result;
}
```

7.3 Bit Operations

```
// Bit an Position j auslesen.
(a & (1 << j)) != 0
// Bit an Position j setzen.
a |= (1 << j)
// Bit an Position j löschen.
a &= ~(1 << j)
// Bit an Position j umkehren.
a ^= (1 << j)
// Wert des niedrigsten gesetzten Bits.
(a & -a)
// Setzt alle Bits auf 1.
a = -1
// Setzt die ersten n Bits auf 1. Achtung: Overflows.
a = (1 << n) - 1
```

7.4 Josephus-Problem

n Personen im Kreis, jeder k-te wird erschossen.

Spezialfall $k = 2$: Betrachte Binärdarstellung von n. Für $n = 1b_1b_2b_3..b_n$ ist $b_1b_2b_3..b_n1$ die Position des letzten Überlebenden. (Rotiere n um eine Stelle nach links)

```cpp
int rotateLeft(int n) { // Der letzte Überlebende, 1-basiert.
  for (int i = 31; i >= 0; i--)
    if (n & (1 << i)) {
      n &= ~(1 << i);
      break;
    }
  n <<= 1; n++; return n;
}
```

Allgemein: Sei $F(n,k)$ die Position des letzten Überlebenden. Nummeriere die Personen mit $0,1,\ldots,n-1$. Nach Erschießen der k-ten Person, hat der Kreis noch Größe $n-1$ und die Position des Überlebenden ist jetzt $F(n-1,k)$. Also: $F(n,k) = (F(n-1,k)+k)\%n$. Basisfall: $F(1,k) = 0$.

```cpp
int josephus(int n, int k) { // Der letzte Überlebende, 0-basiert.
  if (n == 1) return 0;
  return (josephus(n - 1, k) + k) % n;
}
```

Beachte bei der Ausgabe, dass die Personen im ersten Fall von $1,\ldots,n$ nummeriert sind, im zweiten Fall von $0,\ldots,n-1$!

7.5 Gemischtes

- Johnsons *Reweighting Algorithmus:* Füge neue Quelle `s` hinzu, mit Kanten mit Gewicht 0 zu allen Knoten. Nutze Bellmann-Ford zum Betsimmen der Entfernungen `d[i]` von `s` zu allen anderen Knoten. Stoppe, wenn es negative Zyklen gibt. Sonst ändere die gewichte von allen Kanten `(u,v)` im ursprünglichen Graphen zu `d[u]+w[u,v]-d[v]`. Dann sind alle Kantengewichte nichtnegativ, Dijkstra kann angewendet werden.
- Für ein System von Differenzbeschränkungen: Ändere alle Bedingungen in die Form $a - b \leq c$. Für jede Bedingung füge eine Kante `(b,a)` mit Gweicht `c` ein. Füge Quelle `s` hinzu, mit Kanten zu allen Knoten mit Gewicht 0. Nutze Bellmann-Ford, um die kürzesten Pfade von `s` aus zu finden. `d[v]` ist mögliche Lösung für `v`.
- Min-Weight-Vertex-Cover im bipartiten Graph: Partitioniere in `A, B` und füge Kanten `s -> A` mit Gewicht `w(A)` und Kanten `B -> t` mit Gewicht `w(B)` hinzu. Füge Kanten mit Kapazität ∞ von `A` nach `B` hinzu, wo im originalen Graphen Kanten waren. Max-Flow ist die Lösung.
  Im Residualgraphen:
  - Das Vertex-Cover sind die Knoten inzident zu den Brücken. *oder*
  - Die Knoten in `A`, die *nicht* von `s` erreichber sind und die Knoten in `B`, die von `erreichber` sind.
- Allgemeiner Graph: Das Komplement eines Vertex-Cover ist ein Independent Set. $\Rightarrow$ Max Weight Independent Set ist Komplement von Min Weight Vertex Cover.
- Bipartiter Graph: Min Vertex Cover (kleinste Menge Kanten, die alle Knoten berühren) = Max Matching.
- Bipartites Matching mit Gewichten auf linken Knoten. Minimiere Matchinggewicht. Lösung: Sortiere Knoten links aufsteigend nach Gewicht, danach nutze normlen Algorithmus (Kuhn, Seite 7)

7.6 Sonstiges

```cpp
// Alles-Header.
#include <bits/stdc++.h>

// Setzt das deutsche Tastaturlayout.
setxkbmap de

// Schnelle Ein-/Ausgabe mit cin/cout.
ios::sync_with_stdio(false);

// Set mit eigener Sortierfunktion. Typ muss nicht explizit angegeben werden.
set<point2, decltype(comp)> set1(comp);

// PI
#define PI (2*acos(0))

// STL-Debugging, Compiler flags.
-D_GLIBCXX_DEBUG
#define _GLIBCXX_DEBUG

// 128-Bit Integer. Muss zum Einlesen/Ausgeben in einen int oder long long gecastet werden.
__int128
```