Team Contest Reference

ChaosKITs
Karlsruhe Institute of Technology

12. Januar 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 -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]);
}

// Einmal vor allen anderen Operationen aufrufen.
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]);
}
```

`update()` kann so umgeschrieben werden, dass ganze Intervalle geändert werden. Dazu muss ein Offset in den inneren Knoten des Baums gespeichert werden.

1.3 Fenwick Tree

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

//Build an Fenwick-Tree over an array a. Time Complexity: O(n*log(n))
buildFenwickTree(vector<int>& a) {
  n = a.size();
  FT.assign(n+1,0);
  for(int i = 0; i < n; i++)  updateFT(i,a[i]);
}

//Prefix-Sum of intervall [0..i]. Time Complexity: O(log(n))
int prefix_sum(int i) {
  int sum = 0; i++;
  while(i > 0) { sum += FT[i]; i -= (i & (-i)); }
  return sum;
}

//Adds val to index i. Time Complexity O(log(n))
void updateFT(int i, int val) {
  i++; while(i <= n) { FT[i] += val; i += (i & (-i)); }
}
```

1.4 Range Minimum Query

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

//Runtime: 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]);
      }
    }
  }
}
//returns index of minimum! [l, r)
//Runtime: 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 = successor of 10 = min i such that X[i] >= 10
  return 0;
}
```

2 Graphen

2.1 Minimale Spannbäume

Benutze Algorithmus von Kruskal oder Algorithmus von Prim.

Schnitteigenschaft Für jeden Schnitt C im Graphen gilt: Gibt es eine Kante e, die echt leichter ist als alle anderen Schnittkanten, so gehört diese zu allen minimalen Spannbäumen. ($\Rightarrow$ 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

```
typedef pair<int,int> ii;
typedef vector<pair<int,ii>> graph;

//Takes a Graph g (EdgeList!!!) with N nodes and computes the MST and Cost of it. Runtime: O(|E|*log(|E|))
//Requires UnionFind-Datastructure!!!
pair<graph,int> buildMST(int N, graph& g) {
  UnionFind uf(N);
  graph mst; int mst_cost = 0; int M = g.size();
  sort(g.begin(),g.end());
  for(int i = 0; i < M; i++) {
    int u = g[i].second.first, v = g[i].second.second;
    if(uf.findSet(u) != uf.findSet(v)) {
      mst.push_back(g[i]); mst_cost += g[i].first;
      uf.unionSets(u,v);
    }
  }
  return make_pair(mst,mst_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, parent;
  dist.assign(NUM_VERTICES, INF);
  parent.assign(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;

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

      if (nextDist < dist[nextNode]) {
        dist[nextNode] = nextDist; 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|)
struct edge {
  int from; int to; int cost;
  edge () {};
  edge (int from, int to, int cost) : from(from), to(to), cost(cost) {};
};

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 (int j = 0; j < (int)edges.size(); j++) {
      if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
        dist[edges[j].to] = dist[edges[j].from] + edges[j].cost;
        parent[edges[j].to] = edges[j].from;
      }
    }
  }

  // "dist" und "parent" sind korrekte kürzeste Pfade.
  // Folgende Zeilen prüfen nur negative Kreise.
  for (int j = 0; j < (int)edges.size(); j++) {
    if (dist[edges[j].from] + edges[j].cost < dist[edges[j].to]) {
      // Negativer Kreis gefunden.
    }
  }
}
```

2.2.3 Floyd-Warshall-Algorithmus

```
// Laufzeit: O(|V|^3)
// Initialisiere mat: mat[i][i] = 0, mat[i][j] = INF falls i & j nicht adjazent, Länge sonst.
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];
        }
      }
    }
  }
}
```

- Evtl. überschreibt die Eingabe die Nullen auf der Hauptdiagonalen. Nur negative Werte sollten die Nullen überschreiben.
- Von parallelen Kanten sollte nur die günstigste gespeichert werden.
- Knoten `i` liegt genau dann auf einem negativen Kreis, wenn `dist[i][i] < 0` ist.
- Ein u-v-Pfad existiert nicht, wenn `dist[u][v] == INF`.
- Gibt es einen Knoten `c`, sodass `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;
stack<int> s;

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

  for (int i = 0; i < (int)adjlist[v].size(); i++) {
    int u = adjlist[v][i];
    if (!visited[u]) {
      visit(u);
      low[v] = min(low[v], low[u]);
    } else if (inStack[u]) {
      low[v] = min(low[v], low[u]);
    }
  }

  if (d[v] == low[v]) {
    int u;
    do {
      u = s.top(); s.pop(); inStack[u] = false;
      sccs[u] = sccCounter;
    } while(u != v);
    sccCounter++;
  }
}

void scc() {
  // Initialisiere adjlist!
  visited.clear(); visited.assign(NUM_VERTICES, false);
  d.clear(); d.resize(NUM_VERTICES);
  low.clear(); low.resize(NUM_VERTICES);
  inStack.clear(); inStack.assign(NUM_VERTICES, false);
  sccs.clear(); sccs.resize(NUM_VERTICES);

  counter = 0;
  sccCounter = 0;
  for (i = 0; i < NUM_VERTICES; i++) {
    if (!visited[i]) {
      visit(i);
    }
  }
  // sccCounter ist Anzahl der starkem Zusammenhangskomponenten.
  // sccs enthält den Index der SCC für jeden Knoten.
}
```

2.4 Artikulationspunkte und Brücken

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

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

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

```
      if (low[*vit] > maxlow) {
        maxlow = low[*vit];
      }

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

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

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

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

2.5 Eulertouren

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

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

Abbildung 1: Idee für Eulerzyklen

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

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

```
  swap(otherIdx[n][a], otherIdx[n][b]);
  otherIdx[neighA][idxNeighA] = b;
  otherIdx[neighB][idxNeighB] = a;
}

void removeEdge(int n, int i) { // Entfernt Kante i von Knoten n (und die zugehörige Rückwärtskante).
  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! Was ist mit 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 am Ende in cycle (umgekehrte Reihenfolge).
}
```

Achtung:

- Die Ausgabe erfolgt in falscher Reihenfolge.
- Algorithmus schlägt nicht fehl, falls kein Eulerzyklus existiert. Die Existenz muss separat geprüft werden.

2.6 Lowest Common Ancestor

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

//Runtime: O(n)
void initLCA(int gi, int d, int &c) {
  visited[c] = gi, depth[c] = d, first[gi] = min(c, first[gi]), c++;
  for(int gn : graph[gi]) {
    initLCA(gn, d+1, c);
    visited[c] = gi, depth[c] = d, c++;
  }
}
//[a, b]
//Runtime: O(1)
int getLCA(int a, int b) {
  return visited[queryRMQ(min(first[a], first[b]), max(first[a], first[b]))];
}
//=> int c = 0; initLCA(0,0,c); initRMQ(); done! [rmq on 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, kein Einfluss auf die Laufzeit.
```

```
  struct edge { int dest, rev; ll capacity, 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.capacity >= capacity && dfs(e.dest)) {
        e.capacity -= capacity; adjlist[e.dest][e.rev].capacity += 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 der unterschiedlichen Pfade ohne gemeinsame Kanten.
- **Maximum Independent Paths**
  Finde die maximale Anzahl Pfade von s nach t, die keinen Knoten teilen.
  1. Setze s als Quelle, t als Senke und die Kapazität jeder Kante *und jedes Knotens* auf 1.
  2. Der maximale Fluss entspricht der unterschiedlichen Pfade ohne gemeinsame Knoten.
- **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

```
typedef long long ll;
static const ll flowlimit = 1LL << 60; // Should be bigger than the max flow.
struct MinCostFlow { // Should be initialized with new.
  static const int maxn = 400; // Should be bigger than the #vertices.
  static const int maxm = 5000; //#edges.
  struct edge { int node; int next; ll flow; ll value; } edges[maxm << 1];
  int graph[maxn], queue[maxn], pre[maxn], con[maxn], 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 = 0;
      mincost = 0;
      while (SPFA()) {
        extend();
      }
    }
};
```

2.9 Maximal Cardinatlity Bipartite Matching

```
// Laufzeit: O(n*(|V|+|E|))
vector< vector<int> > adjlist; // Gerichtete Kanten, von links nach rechts.
vector<int> pairs; // Zu jedem Knoten der gematchte Knoten rechts, oder -1.
vector<bool> visited;

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

// n = #Knoten links (0..n-1), m = #Knoten rechts
int kuhn(int n, int m) {
  pairs.assign(n + m, -1);
  int ans = 0;
  for (int i = 0; i < n; i++) {
    visited.assign(n + m, false);
    ans += dfs(i);
  }
  return ans; // Größe des Matchings.
}
```

2.10 TSP

```
// Laufzeit: O(n*2^n)
// nodes[0] ist Start- und Endknoten.
vector<vector<int>> dist;
vector<int> TSP() {
  int n = dist.size(), m = 1 << n;
  vector<vector<ii>> dp(n, vector<ii>(m, ii(MAX_N, -1)));

  for(int c = 0; c < n; c++) dp[c][m-1].first = dist[c][0], dp[c][m-1].second = 0;

  for(int v = m - 2; v >= 0; v--) {
    for(int c = n - 1; c >= 0; c--) {
      for(int g = 0; g < n; g++) {
        if(g != c && (((1 << g) & v) == 0)) {
          if((dp[g][(v | (1 << g))].first + dist[c][g]) < dp[c][v].first) {
            dp[c][v].first = dp[g][(v | (1 << g))].first + dist[c][g];
            dp[c][v].second = g;
          }
        }
      }
    }
  }

  vector<int> res; res.push_back(0); int v = 0;
  while(res.back() != 0 || res.size() == 1) {
    res.push_back(dp[res.back()][(v |= (1 << res.back()))].second);
  }

  return res;
}
```

2.11 Bitonic TSP

```
// Laufzeit: O(|V|^2)
vector< vector<double> > dp; // Initialisiere mit -1
vector< vector<double> > dist; // Initialisiere mit Entfernungen zwischen Punkten.
vector<int> lr, rl; // Links-nach-rechts und rechts-nach-links Pfade.
int n; // #Knoten
```

```

// get(0, 0) gibt die Länge der kürzesten bitonischen Route.
double get(int p1, int p2) {
  int v = max(p1, p2) + 1;
  if (v == n - 1) return dist[p1][v] + dist[v][p2];
  if (dp[p1][p2] > -0.5) return dp[p1][p2];
  double tryLR = dist[p1][v] + get(v, p2), tryRl = dist[v][p2] + get(p1, v);
  if (tryLR < tryRL) lr.push_back(v); // Baut die Pfade auf. Fügt v zu rl hinzu, falls beide gleich teuer.
  else rl.push_back(v); // Änder das, falls nötig.
  return min(tryLR, tryRL);
}
```

3 Geometrie

3.1 Closest Pair

```
double squaredDist(point a, point b) {
  return (a.first-b.first) * (a.first-b.first) + (a.second-b.second) * (a.second-b.second);
}

bool compY(point a, point b) {
  if (a.second == b.second) return a.first < b.first;
  return a.second < b.second;
}

double shortestDist(vector<point> &points) {
  //check that points.size() > 1 and that ALL POINTS ARE DIFFERENT
  set<point, bool(*)(point, point)> status(compY);
  sort(points.begin(), points.end());
  double opt = 1e30, sqrtOpt = 1e15;
  auto left = points.begin(), right = points.begin();
  status.insert(*right); right++;

  while (right != points.end()) {
    if (fabs(left->first - right->first) >= sqrtOpt) {
      status.erase(*(left++));
    } else {
      auto lower = status.lower_bound(point(-1e20, right->second - sqrtOpt));
      auto upper = status.upper_bound(point(-1e20, right->second + sqrtOpt));
      while (lower != upper) {
        double cand = squaredDist(*right, *lower);
        if (cand < opt) {
          opt = cand;
          sqrtOpt = sqrt(opt);
        }
        ++lower;
      }
      status.insert(*(right++));
    }
  }
  return sqrtOpt;
}
```

3.2 Geraden

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

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

```

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

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

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

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

3.3 Konvexe Hülle

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

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

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

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

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

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

```

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

3.4 Formeln - std::complex

```
//komplexe Zahlen als Darstellung fuer Punkte
typedef pt complex<double>;
//Winkel zwischen Punkt und x-Achse in [0, 2 * PI), Winkel zwischen a und b
double angle = arg (a), angle_a_b = arg (a - b);
//Punkt rotiert um Winkel theta
pt a_rotated = a * exp (pt (0, theta));
//Mittelpunkt des Dreiecks abc
pt centroid = (a + b + c) / 3;
//Skalarprodukt
double dot(pt a, pt b) {
  return real(conj(a) * b);
}
//Kreuzprodukt, 0, falls kollinear
double cross(pt a, pt b) {
  return imag(conj(a) * b);
}
//wenn Eckpunkte bekannt
double areaOfTriangle(pt a, pt b, pt c) {
  return abs(cross(b - a, c - a)) / 2.0;
}
//wenn Seitenlaengen bekannt
double areaOfTriangle(double a, double b, double c) {
  double s = (a + b + c) / 2;
  return sqrt(s * (s-a) * (s-b) * (s-c));
}
// Sind die Dreiecke a1, b1, c1, and a2, b2, c2 aehnlich?
// Erste Zeile testet Aehnlichkeit mit gleicher Orientierung,
// zweite Zeile testst Aehnlichkeit mit unterschiedlicher Orientierung
bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
  return (
    (b2 - a2) * (c1 - a1) == (b1 - a1) * (c2 - a2) ||
    (b2 - a2) * (conj (c1) - conj (a1)) == (conj (b1) - conj (a1)) * (c2 - a2)
  );
}
//Linksknick von a->b nach a->c
double ccw(pt a, pt b, pt c) {
  return cross(b - a, c - a); //<0 => falls Rechtsknick, 0 => kollinear, >0 => Linksknick
}
//Streckenschnitt, Strecken a-b und c-d
bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
  if (ccw(a, b, c) == 0 && ccw(a, b, d) == 0) { //kollinear
    double dist = abs(a - b);
    return (abs(a - c) <= dist && abs(b - c) <= dist) || (abs(a - d) <= dist && abs(b - d) <= dist);
  }
  return ccw(a, b, c) * ccw(a, b, d) <= 0 && ccw(c, d, a) * ccw(c, d, b) <= 0;
}
//Entfernung von p zu a-b
double distToLine(pt a, pt b, pt p) {
  return abs(cross(p - a, b - a)) / abs(b - a);
}
//liegt p auf a-b
bool pointOnLine(pt a, pt b, pt p) {
  return abs(distToLine(a, b, p)) < EPSILON;
}
//testet, ob d in der gleichen Ebene liegt wie a, b, und c
bool isCoplanar(pt a, pt b, pt c, pt d) {
  return (b - a) * (c - a) * (d - a) == 0;
}
//berechnet den Flaecheninhalt eines Polygons (nicht selbstschneidend)
double areaOfPolygon(vector<pt> &polygon) { //jeder Eckpunkt nur einmal im Vektor
  double res = 0; int n = polygon.size();
  for (int i = 0; i < (int)polygon.size(); i++)
```

```
    res += real(polygon[i]) * imag(polygon[(i + 1) % n]) - real(polygon[(i + 1) % n]) * imag(polygon[i]);
  return 0.5 * abs(res);
}
//testet, ob sich zwei Rechtecke (p1, p2) und (p3, p4) schneiden (jeweils gegenueberliegende Ecken)
bool rectIntersection(pt p1, pt p2, pt p3, pt p4) {
  double minx12 = min(real(p1), real(p2)), maxx12 = max(real(p1), real(p2));
  double minx34 = min(real(p3), real(p4)), maxx34 = max(real(p3), real(p4));
  double miny12 = min(imag(p1), imag(p2)), maxy12 = max(imag(p1), imag(p2));
  double miny34 = min(imag(p3), imag(p4)), maxy34 = max(imag(p3), imag(p4));
  return (maxx12 >= minx34) && (maxx34 >= minx12) && (maxy12 >= miny34) && (maxy34 >= miny12);
}
//testet, ob ein Punkt im Polygon liegt (beliebige Polygone)
bool pointInPolygon(pt p, vector<pt> &polygon) { //jeder Eckpunkt nur einmal im Vektor
  pt rayEnd = p + pt(1, 1000000);
  int counter = 0, n = polygon.size();
  for (int i = 0; i < n; i++) {
    pt start = polygon[i], end = polygon[(i + 1) % n];
    if (lineSegmentIntersection(p, rayEnd, start, end)) counter++;
  }
  return counter & 1;
}
```

4 Mathe

4.1 ggT, kgV, erweiterter euklidischer Algorithmus

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

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

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

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

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

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

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

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

4.2 Primzahlsieb von Eratosthenes

```
#define N 10000001
vector<ll> primes;
//Finds all prime numbers between 0..N
//Use this method for N < 10000000 to avoid memory access errors
void primeSieve() {
  bitset<N> isPrime; isPrime.set();
  isPrime[0] = isPrime[1] = 0;
  for(ll i = 2; i < N+1; i+=2) {
      if(isPrime[i]) {
        for(ll j = i*i; j >= 0 && j < N+1; j+=i) isPrime[j] = 0;
        primes.push_back(i);
      }
      if(i == 2) i--;
  }
}
```

4.2.1 Faktorisierung

```
typedef pair<int,int> ii;
//Factorize a number n in its prime factors
//Call primeSieve-method before with N > sqrt(n)
//Return: Returns a vector of pairs, where the first entry in the pair is
//the prime factor p and the second counts how many times p divides n
vector<ii> factorize(ll n) {
  vector<ii> fact; ll num = n, i = 0, c = 0;
  while(num != 1) {
    if(num % primes[i] == 0) {
      c++; num /= primes[i];
    } else {
      if(c > 0)
        fact.push_back(make_pair(primes[i],c));
      i++; c = 0;
      if(primes[i]*primes[i] > num) break;
    }
  }
  if(num != 1) fact.push_back(make_pair(num,c+1));
  return fact;
}
```

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

```
//0<=a,b <=n and n <= MAX(ll)/2
ll mult_mod(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 + mult_mod(a, b-1, n)) % n;
  else return mult_mod((a + a) % n, b / 2, n);
}

//0<=a,b<=n and n <= MAX(ll)/2
ll pow_mod(ll a, ll b, ll n) {
  if(b == 0) return 1;
  if(b == 1) return a % n;

  if(b % 2 == 1) return mult_mod(pow_mod(a, b-1, n), a, n);
  else return pow_mod(mult_mod(a, a, n), b / 2, n);
}
```

4.2.3 LGS über $\mathbb{F}_p$

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

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

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

4.3 Binomialkoeffizienten

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

4.4 Satz von Sprague-Grundy

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

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

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

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

```
#Most important function!!!11elf
bool WinNimm(vector<int> game) {
  int result = 0;
  for(int s: game) result ^= s;
  return s > 0;
}
```

4.5 Maximales Teilfeld

```
//N :=  length of field
int maxStart = 1, maxLen = 0, curStart = 1, len = 0;
double maxValue = 0, sum = 0;
for (int pos = 0; pos < N; pos++) {
```

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

Obiger Code findet kein maximales Teilfeld, das über das Ende hinausgeht. Dazu:

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

4.6 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]))`

```
// s.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_1, ... & b_0, b_1, ...
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
```

4.7 Kombinatorik

4.7.1 Berühmte Zahlen

Fibonacci-Zahlen	$f(0) = 0 \quad f(1) = 1 \quad f(n+2) = f(n+1) + f(n)$	Bem. 1, 2
Catalan-Zahlen	$C_0 = 1 \quad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1}\binom{2n}{n} = \frac{2(2n-1)}{n+1} \cdot C_{n-1}$	Bem. 3, 4
Euler-Zahlen (I)	$\left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1 \quad \left\langle {n \atop k} \right\rangle = (k+1)\left\langle {n-1 \atop k} \right\rangle + (n-k)\left\langle {n-1 \atop k-1} \right\rangle$	Bem. 5
Euler-Zahlen (II)	$\left\langle\!\left\langle {n \atop 0} \right\rangle\!\right\rangle = 1 \quad \left\langle\!\left\langle {n \atop n} \right\rangle\!\right\rangle = 0 \quad \left\langle\!\left\langle {n \atop k} \right\rangle\!\right\rangle = (k+1)\left\langle\!\left\langle {n-1 \atop k} \right\rangle\!\right\rangle + (2n-k-1)\left\langle\!\left\langle {n-1 \atop k-1} \right\rangle\!\right\rangle$	Bem. 6
Stirling-Zahlen (I)	$\left[{0 \atop 0} \right] = 1 \quad \left[{n \atop 0} \right] = \left[{0 \atop n} \right] = 0 \quad \left[{n \atop k} \right] = \left[{n-1 \atop k-1} \right] + (n-1)\left[{n-1 \atop k} \right]$	Bem. 7
Stirling-Zahlen (II)	$\left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1 \quad \left\{ {n \atop k} \right\} = k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}$	Bem. 8
Integer-Partitions	$f(1,1) = 1 \quad f(n,k) = 0$ für $k > n \quad f(n,k) = f(n-k,k) + f(n,k-1)$	Bem. 9

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 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 in einem* $n \times n$*-Gitter, die nicht die Diagonale kreuzen. (zwischen gegenüberliegenden Ecken)*

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

Bemerkung 7 (Stirling-Zahlen 1. Ordnung) *Die Anzahl der Permutationen von* $\{1, \ldots, n\}$ *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* $\leq k$*.*

4.7.2 Verschiedenes

Hanoi Towers (min steps)	$T_n = 2^n - 1$
#regions by n lines	$n(n+1)/2 + 1$
#bounded regions by n lines	$\left(n^2 - 3n + 2\right)/2$
#labeled rooted trees	n^{n-1}
#labeled unrooted trees	n^{n-2}

5 Strings

5.1 Knuth-Morris-Pratt-Algorithmus

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

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

5.2 Levenshtein-Distanz

```
int levenshtein(string& s1, string& s2) {
  int len1 = s1.size(), len2 = s2.size();
  vector<int> col(len2 + 1), prevCol(len2 + 1);
  for (int i = 0; i < len2 + 1; i++) prevCol[i] = i;
  for (int i = 0; i < len1; i++) {
    col[0] = i + 1;
    for (int j = 0; j < len2; j++)
      col[j+1] = min(min(prevCol[j+1] + 1, col[j] + 1), prevCol[j] + (s1[i]==s2[j] ? 0 : 1));
    col.swap(prevCol);
  }
  return prevCol[len2];
}
```

5.3 Trie

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

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

```
    }
    insert(root->e[idx], txt, s+1);
  }
}

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

5.4 Suffix-Array

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

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

5.5 Longest Common Substring

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

```
      if(s[u.s][ui] < s[l.s][li]) return true;
      else if(s[u.s][ui] > s[l.s][li]) return false;
      ui++; li++;
    }
    return !(ui < s[u.s].length());
  });
  int r = 0, m=0, c=0;
  for(int i = 0; i < a.size() - 1; i++) {
    if(a[i].s == a[i+1].s) continue;
    c = 0;
    while(a[i].i+c < s[a[i].s].length() && a[i+1].i+c < s[a[i+1].s].length() && s[a[i].s][a[i].i+c] == s[a[i
         +1].s][a[i+1].i+c]) c++;
    if(c > m) r=i, m=c;
  }
  return m == 0 ? "" : s[a[r].s].substr(a[r].i, m);
}
```

5.6 Longest Common Subsequence

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

6 Java

6.1 Introduction

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

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

- Ausgabe:

```
//Ausgabe in StringBuilder schreiben und am Ende alles auf einmal ausgeben -> viel schneller
StringBuilder sb = new StringBuilder(); //java.lang.StringBuilder
sb.append("Hallo Welt");
System.out.print(sb.toString());
```

6.2 BigInteger

Hier ein kleiner überblick über die Methoden der Klasse BigInteger:

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

```

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

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

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

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

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

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

7 Sonstiges

7.1 2-SAT

1. Bedingungen in 2-CNF formulieren.
2. Implikationsgraph bauen, $(a \lor b)$ 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 Sortieren in Linearzeit

Wenn die Eingabe aus einem kleinen Intervall $[0, n)$ stammt ist Bucketsort schneller.

7.2.1 Bucketsort

```
vector<int> res;
void bucketSort(vector<int> &a) { //stores result in global vector res
  int c[BUCKETS] = {0};
  for (int i = 0; i < (int)a.size(); i++) c[a[i]]++;
  int C = 0;
  for (int i = 0; i < BUCKETS; i++) {
    int tmp = C;
    C += c[i];
    c[i] = tmp;
  }
  res.resize(a.size());
  for (int i = 0; i < (int)a.size(); i++) {
    res[c[a[i]]] = a[i];
    c[a[i]]++;
  }
}
```

7.2.2 LSD-Radixsort

```
//Comparable with sort from <algorithms> in a range from 0 to 5000, for values greater than 5000 use sort
const int p[10] = {1,10,100,1000,10000,100000,1000000,10000000,100000000,1000000000};

int getLongestNumber(vector<int> &a) {
  int res = 0;
  for (int i = 0; i < (int)a.size(); i++) res = max(res, (int)ceil(log10(a[i]) + 1));
  return res;
}

```

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

void radixSort(vector<int> &a) {
  int digits = getLongestNumber(a);
  for (int d = 0; d < digits; d++) {
    vector<int> bucket[10];
    for(int i = 0; i < (int)a.size(); i++)
      bucket[getIthDigit(a[i],d)].push_back(a[i]);
    a.clear();
    for(int i = 0; i < 10; i++)
      copy(bucket[i].begin(), bucket[i].end(), back_inserter(a));
  }
}
```

7.3 Bit Operations

```
//lsb: 0-th bit, msb: n-th bit
//get j-th bit
(a & (1 << j)) != 0
//set j-th bit
a |= (1 << j)
//clear j-th bit
a &= ~(1 << j)
//toggle j-th bit
a ^= (1 << j)
//get value of least significant bit set
(a & -a)
//turn on all bits
a = -1
//turn on first n bits (be aware of 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)

```
int rotateLeft(int n) { //returns the number of the last survivor (1 based)
  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 $0,1,\dots,n-1$. 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: $F(1,k) = 0$.

```
int josephus(int n, int k) { //returns the number of the last survivor (0 based)
  if (n == 1) return 0;
  return (josephus(n - 1, k) + k) % n;
}
```

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

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 11)

- **Tobi, cool down!**

8 Convenience-Methoden

8.1 Zeileneingabe

```
vector<string> split(string &s, string delim) { //zerlegt s anhand aller Zeichen in 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;
}
```

8.2 Template

```
#include <bits/stdc++.h>

using namespace std;

void solve() {}

int main() {
  solve();
  return 0;
}
```

8.3 Deutsches Tatstaturlayout

```
setxkbmap de
```