Team Contest Reference

Hello KITty
Karlsruhe Institute of Technology

20. November 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;
}
```

1.6 STL-Rope (Implicit Cartesian Tree)

```
#include <ext/rope>
using namespace __gnu_cxx;
rope<int> v; // Wie normaler Container.
v.push_back(num); // O(log(n))
rope<int> sub = v.substr(start, length); // O(log(n))
v.erase(start, length); // O(log(n))
v.insert(v.mutable_begin() + offset, sub); // O(log(n))
for(auto it = v.mutable_begin(); it != v.mutable_end(); it++) {...}
```

1.7 Treap (Cartesian Tree)

```
struct item {
  int key, prior;
  item *l, *r;
  item() {}
  item(int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) {}
};

void split(item *t, int key, item *l, item *r) {
  if (!t) l = r = NULL;
  else if (key < t->key) split(t->l, key, l, t->l), r = t;
  else split(t->r, key, t->r, r), l = t;
}

void insert(item *t, item *it) {
  if (!t) t = it;
  else if (it->prior > t->prior) split(t, it->key, it->l, it->r), t = it;
  else insert(it->key < t->key ? t->l : t->r, it);
}

void merge(item *t, item *l, item *r) {
  if (!l || !r) t = l ? l : r;
  else if (l->prior > r->prior) merge(l->r, l->r, r), t = l;
  else merge(r->l, l, r->l), t = r;
}

void erase(item *t, int key) {
  if (t->key == key) merge (t, t->l, t->r);
  else erase(key < t->key ? t->l : t->r, key);
}

item *unite(item *l, item *r) {
  if (!l || !r) return l ? l : r;
  if (l->prior < r->prior) swap(l, r);
  item * lt, rt;
  split(r, l->key, lt, rt);
  l->l = unite(l->l, lt);
  l->r = unite(l->r, rt);
  return l;
}
```

1.8 Erste unbenutzte natürliche Zahl

```
// Erste natürliche Zahl nicht im set used.
set<int> used;
int unusedCounter = 1;

int getFirstUnused() { // Laufzeit: O(log n) amortisiert.
  auto it = used.lower_bound(unusedCounter);
  while (it != used.end() && *it == unusedCounter) {
    it++;
    unusedCounter++;
  }
  return unusedCounter++; // Evtl. neuen Wert noch hinzufügen.
}
```

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. (⇒ 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, rootCount; // rootCount >= 2 <=> root Artikulationspunkt
vector<ii> bridges; // Nur fuer Brücken.

void dfs(int v, int parent = -1) {
  d[v] = low[v] = ++counter;
  if (parent == root) ++rootCount;

  for (auto w : adjlist[v]) {
    if (!d[w]) {
      dfs(w, v);
      if (low[w] >= d[v] && v != root) 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() {
  counter = 0;
  low.resize(adjlist.size());
  d.assign(adjlist.size(), 0);
  isArt.assign(adjlist.size(), false);
  bridges.clear(); //nur fuer Bruecken
  for (int v = 0; v < (int)adjlist.size(); v++) {
    if (!d[v]) {
      root = v; rootCount = 0;
      dfs(v);
      if (rootCount > 1) isArt[v] = true;
}}}
```

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);

// Funktioniert nur mit von der Wurzel weggerichteten Kanten.
// Falls ungerichtete Kanten, visited-check einführen.
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))
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], target, dfsCounter;
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 Dinic's Algorithm mit Capacity Scaling

Nochmal ca. Faktor 2 schneller als Ford Fulkerson mit Capacity Scaling.

```
// Laufzeit: O(|V|^2*|E|)
// Knoten müssen von 0 nummeriert sein.
const int INF = 0x3FFFFFFF, MAXN = 500;
struct edge { int a, b; ll f, c; };
int n, m, pt[MAXN], d[MAXN], s, t;
vector<edge> e;
vector<int> g[MAXN];
ll flow = 0, lim;
queue<int> q;

void addEdge(int a, int b, ll c) {
  g[a].push_back(e.size());
  e.push_back(edge {a, b, 0, c});
  g[b].push_back(e.size());
  e.push_back(edge {b, a, 0, 0});
}

bool bfs() {
  for (int i = 0; i < n; i++) d[i] = INF;
  d[s] = 0;
  q.push(s);
  while (!q.empty() && d[t] == INF) {
    int cur = q.front(); q.pop();
    for (int i = 0; i < (int)g[cur].size(); i++) {
      int id = g[cur][i], to = e[id].b;
      if (d[to] == INF && e[id].c - e[id].f >= lim) {
        d[to] = d[cur] + 1;
        q.push(to);
      }
    }
  }
  while (!q.empty()) q.pop();
  return d[t] != INF;
}

bool dfs(int v, ll flow) {
  if (flow == 0) return false;
  if (v == t) return true;
  for (; pt[v] < (int)g[v].size(); pt[v]++) {
    int id = g[v][pt[v]], to = e[id].b;
```

```
    if (d[to] == d[v] + 1 && e[id].c - e[id].f >= flow) {
      int pushed = dfs(to, flow);
      if (pushed) {
        e[id].f += flow;
        e[id ^ 1].f -= flow;
        return true;
      }
    }
  }
  return false;
}

// Nicht vergessen, s und t zu setzen!
void dinic() {
  for (lim = (1LL << 62); lim >= 1;) {
    if (!bfs()) { lim /= 2; continue; }
    for (int i = 0; i < n; i++) pt[i] = 0;
    int pushed;
    while ((pushed = dfs(s, lim))) flow += lim;
  }
}
```

2.7.4 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 Anzahl 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|))
// Kanten von links nach rechts. Die ersten n Knoten sind links, die
    anderen rechts.
vector<vector<int>> adjlist;
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.
}
```

```
// Laufzeit: O(sqrt(|V|)*|E|)
// Kanten von links nach rechts.
// 0: dummy Knoten, 1..n: linke Knoten, n+1..n+m: rechte Knoten
vector<vector<int>> adjlist;
vector<int> match, dist;

bool bfs(int n) {
  queue<int> q;
  dist[0] = INF;
  for(int i = 1; i <= n; i++) {
    if(match[i] == 0) { dist[i] = 0; q.push(i); }
    else dist[i] = INF;
  }
  while(!q.empty()) {
    int u = q.front(); q.pop();
    if(dist[u] < dist[0]) for (int v : adjlist[u])
      if(dist[match[v]] == INF) {
        dist[match[v]] = dist[u] + 1;
        q.push(match[v]);
      }
  }
  return dist[0] != INF;
}

bool dfs(int u) {
  if(u != 0) {
    for (int v : adjlist[u])
      if(dist[match[v]] == dist[u] + 1)
        if(dfs(match[v])) { match[v] = u; match[u] = v; return true; }
    dist[u] = INF;
    return false;
  }
  return true;
}

int hopcroft_karp(int n) { // n = #Knoten links
  int ans = 0;
  match.assign(adjlist.size(), 0);
  dist.resize(adjlist.size());
  // Greedy Matching, optionale Beschleunigung.
  for (int i = 1; i <= n; i++) for (int w : adjlist[i])
    if (match[w] == 0) { match[i] = w; match[w] = i; ans++; break; }
  while(bfs(n)) for(int i = 1; i <= n; i++)
    if(match[i] == 0 && dfs(i)) ans++;
  return ans;
}
```

2.10 2-SAT

```
struct sat2 {
  vector<vector<int>> adjlist, sccs;
  vector<bool> visited, inStack;
  int n, sccCounter, dfsCounter;
  vector<int> d, low, idx, sol;
  stack<int> s;

  sat2(int vars) : n(vars*2) { adjlist.resize(n); };

  static int var(int i) { return i << 1; }

  void addImpl(int v1, int v2) {
    adjlist[v1].push_back(v2);
    adjlist[1^v2].push_back(1^v1);
  }
  void addEquiv(int v1, int v2) { addImpl(v1, v2); addImpl(v2, v1); }
  void addOr(int v1, int v2) { addImpl(1^v1, v2); }
```

```
  void addXor(int v1, int v2) { addOr(v1, v2); addOr(1^v1, 1^v2); }
  void addTrue(int v1) { addImpl(1^v1, v1); }
  void addFalse(int v1) { addTrue(1^v1); }
  void addAnd(int v1, int v2) { addTrue(v1); addTrue(v2); }
  void addNand(int v1, int v2) { addOr(1^v1, 1^v2); }

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

    for (auto w : adjlist[v]) {
      if (!visited[w]) {
        dfs(w);
        low[v] = min(low[v], low[w]);
      } else if (inStack[w]) low[v] = min(low[v], low[w]);
    }

    if (d[v] == low[v]) {
      sccs.push_back(vector<int>());
      int w;
      do {
        w = s.top(); s.pop(); inStack[w] = false;
        idx[w] = sccCounter;
        sccs[sccCounter].push_back(w);
      } while (w != v);
      sccCounter++;
  }}

  bool solvable() {
    visited.assign(n, false);
    inStack.assign(n, false);
    d.assign(n, -1);
    low.assign(n, -1);
    idx.assign(n, -1);
    sccCounter = dfsCounter = 0;
    for (int i = 0; i < n; i++) if (!visited[i]) dfs(i);
    for (int i = 0; i < n; i += 2) if (idx[i] == idx[i + 1]) return false;
    return true;
  }

  void assign() {
    sol.assign(n, -1);
    for (int i = 0; i < sccCounter; i++) {
      if (sol[sccs[i][0]] == -1) {
        for (int v : sccs[i]) {
          sol[v] = 1;
          sol[1^v] = 0;
  }}}}
};
```

2.11 Bitonic TSP

```
// Laufzeit: O(n^2)
vector<vector<double>> dist; // Initialisiere mit Entfernungen zwischen Punkten.
vector<vector<double>> dp;

double get(int p1, int p2) {
    int v = max(p1, p2) + 1;
    if (v == dist.size()) return dist[p1][v - 1] + dist[p2][v - 1];
    if (dp[p1][p2] >= 0.0) return dp[p1][p2];
    double tryLR = dist[p1][v] + get(v, p2), tryRL = dist[p2][v] + get(p1, v);
  return dp[p1][p2] = min(tryLR, tryRL);
}

void bitonicTour() {
    dp = vector<vector<double>>(dist.size(), vector<double>(dist.size(), -1));
    get(0, 0);
    // return dp[0][0]; // Länger der Tour
    vector<int> lr = {0}, rl = {0};
    for (int p1 = 0, p2 = 0, v; (v = max(p1, p2) + 1) < dist.size();) {
        if (dp[p1][p2] == dist[p1][v] + dp[v][p2]) {
            lr.push_back(v); p1 = v;
        } else {
            rl.push_back(v); p2 = v;
  }}
    lr.insert(lr.end(), rl.rbegin(), rl.rend());
    // return lr; // Enthält Knoten 0 zweimal. An erster und letzter Position.
}
```

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 complex<double> pt;

// Winkel zwischen Punkt und x-Achse in [0, 2 * PI).
double angle = arg(a);

// 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, p) == 0;
}

// Liegt p auf der Strecke a-b?
bool pointOnLineSegment(pt a, pt b, pt p) {
  if (orientation(a, b, p) != 0) return false;
  return real(p) >= min(real(a), real(b)) &&
         real(p) <= max(real(a), real(b)) &&
         imag(p) >= min(imag(a), imag(b)) &&
         imag(p) <= max(imag(a), imag(b));
}

// Entfernung von Punkt p zur Strecke a-b.
double distToSegment(pt a, pt b, pt p) {
  if (a == b) return abs(p - a);
  double segLength = abs(a - b);
  double u = ((real(p) - real(a)) * (real(b) - real(a)) +
      (imag(p) - imag(a)) * (imag(b) - imag(a))) /
      (segLength * segLength);
  pt projection(real(a) + u * (real(b) - real(a)),
      imag(a) + u * (imag(b) - imag(a)));
  double projectionDist = abs(p - projection);
  if (!pointOnLineSegment(a, b, projection)) projectionDist = 1e30;
  return min(projectionDist, min(abs(p - a), abs(p - b)));
}

// Kürzeste Entfernung zwischen den Strecken a-b und c-d.
double distBetweenSegments(pt a, pt b, pt c, pt d) {
  if (lineSegmentIntersection(a, b, c, d)) return 0.0;
  double result = distToSegment(a, b, c);
  result = min(result, distToSegment(a, b, d));
  result = min(result, distToSegment(c, d, a));
  return min(result, distToSegment(c, d, b));
}
```

```cpp

// Liegt d in der gleichen Ebene wie a, b, und c?
bool isCoplanar(pt a, pt b, pt c, pt d) {
  return abs((b - a) * (c - a) * (d - a)) < EPSILON;
}

// Berechnet den Flächeninhalt eines Polygons (nicht selbstschneidend).
// Punkte gegen den Uhrzeigersinn: positiv, sonst negativ.
double areaOfPolygon(vector<pt> &polygon) { // Jeder Eckpunkt nur einmal.
  double res = 0; int n = polygon.size();
  for (int i = 0; i < n; i++)
    res += real(polygon[i]) * imag(polygon[(i + 1) % n]) -
           real(polygon[(i + 1) % n]) * imag(polygon[i]);
  return 0.5 * res;
}

// Schneiden sich (p1, p2) und (p3, p4) (gegenüberliegende 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) { // Punkte nur einmal.
  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

```cpp
// Laufzeiten: O(log(a) + log(b))
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)); }
```

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

Lemma von Bézout Sei (x, y) eine Lösung für $ax + by = d$. Dann lassen sich wie folgt alle Lösungen berechnen:

$$\left(x + k\frac{b}{\mathrm{ggT}(a,b)},\ y - k\frac{a}{\mathrm{ggT}(a,b)}\right)$$

Multiplikatives Inverses von x in $\mathbb{Z}/n\mathbb{Z}$ Sei $0 \le x < n$. Definiere $d := \mathrm{ggT}(x, n)$.

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}.

```cpp
// Laufzeit: O(log (n) + log(p))
ll multInv(ll n, ll p) {
  ll x, y;
  extendedEuclid(n, p, x, y); // Implementierung von oben.
  x = ((x % p) + p) % p;
  return x % p;
}
```

4.2 Mod-Exponent über $\mathbb{F}_p$

```cpp
// 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 & 1) return (powMod(a, b - 1, n) * a) % n;
  else return powMod((a * a) % n, b / 2, n);
}
```

Iterativ:

```cpp
// Laufzeit: O(log (b))
ll powMod(ll a, ll b, ll n) {
  if (b == 0) return 1;
  ll res = 1;
  while (b > 1) {
    if (b & 1) res = (a * res) % n;
    a = (a * a) % n;
    b /= 2;
  }
  return (a * res) % n;
}
```

4.3 Chinesischer Restsatz

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

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

  Formel kann auch für nicht teilerfremde Moduli verwendet werden. Sind die Moduli nicht teilerfremd, existiert genau dann eine Lösung, wenn $a \equiv b \mod \text{ggT}(m, n)$. 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;
  }
};
```

4.4 Primzahltest & Faktorisierung

```
bool isPrime(ll n) { // Miller Rabin Primzahltest. O(log 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 = (v * v) % n;
      if(v == n - 1 || v <= 1) break;
    }
    if(v != n - 1) return false;
  }
  return true;
}

ll rho(ll n) { // Findet Faktor < n, nicht unbedingt prim.
  if (~n & 1) return 2;
  ll c = rand() % n, x = rand() % n, y = x, d = 1;
  while (d == 1) {
    x = ((x * x) % n + c) % n;
    y = ((y * y) % n + c) % n;
    d = gcd(abs(x - y), n); // Implementierung von oben.
  }
  return d == n ? rho(n) : d;
}

void factor(ll n, map<ll, int> &facts) {
  if (n == 1) return;
  if (isPrime(n)) {
    facts[n]++;
    return;
  }
  ll f = rho(n);
  factor(n / f, facts);
  factor(f, facts);
}
```

4.5 Primzahlsieb von Eratosthenes

```
// Laufzeit: O(n * log log n)
// Kann erweitert werden: Für jede Zahl den kleinsten Primfaktor.
// Dabei vorsicht: Nicht kleinere Faktoren überschreiben.
#define N 100000000 // Bis 10^8 in unter 64MB Speicher.
bitset<N / 2> isNotPrime;

inline bool isPrime(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 !isNotPrime[x / 2];
```

```cpp
}

inline int primeSieve() { // Rückgabe: Anzahl der Primzahlen < N.
  int counter = 1; // Die 2, die sonst vergessen würde.
  for (int i = 3; i < N; i += 2) {
    if (!isNotPrime[i / 2]) {
      for (int j = i * i; j < N; j+= 2 * i) isNotPrime[j / 2] = 1;
      counter++;
  }}
  return counter;
}
```

4.6 Eulersche φ-Funktion

- Zählt die relativ primen Zahlen $\leq n$.
- Multiplikativ: $\gcd(a, b) = 1 \Longrightarrow \varphi(a) \cdot \varphi(b) = \varphi(ab)$
- p prim, $k \in \mathbb{N}$: $\varphi(p^k) = p^k - p^{k-1}$
- $n = p_1^{a_1} \cdot \ldots \cdot p_k^{a_k}$: $\varphi(n) = n \cdot \left(1 - \frac{1}{p_1}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_k}\right)$ Evtl. ist es sinnvoll obgien Code zum Faktorisieren zu benutzen und dann diese Formel anzuwenden.
- **Euler's Theorem:** Seien a und m teilerfremd. Dann: $a^{\varphi(m)} \equiv 1 \mod m$
  Falls m prim ist, liefert das den **kleinen Satz von Fermat**: $a^m \equiv a \mod m$

```cpp
ll phi(ll n) { // Laufzeit: O(sqrt(n))
  // Optimierung: Falls n prim, n - 1 zurückgeben (Miller-Rabin/Sieb).
  ll result = n;
  for(int i = 2; i * i <= n; ++i) {
    if(n % i == 0) {  // Optimierung: Nur über Primzahlen iterieren.
      while(n % i == 0)n /= i;
      result -= result / i;
    }
  }
  if(n > 1) result -= result / n;
  return result;
}
```

4.7 Primitivwurzeln

- Primitivwurzel modulo n existiert genau dann wenn:
  - n ist 1, 2 oder 4, oder
  - n ist Potenz einer ungeraden Primzahl, oder
  - n ist das Doppelte einer Potenz einer ungeraden Primzahl.
- Sei g Primitivwurzel modulo n. Dann gilt:
  Das kleinste k, sodass $g^k \equiv 1 \mod n$, ist $k = \varphi(n)$.

```cpp
// Ist g Primitivwurzel modulo p. Teste zufällige g, um eine zu finden.
bool is_primitive(ll g, ll p) {
  map<ll, int> facs;
  factor(p - 1, facs);
  for (auto &f : facs)
    if (1 == powMod(g, (p - 1) / f.first, p)) return false;
  return true;
}

// Alternativ: Generator zum Finden. -1 falls keine existiert.
ll generator (ll p) { // Laufzeit: O(ans*log(phi(n))*log(n))
  map<ll, int> facs;
  factor(n, facs);
  ll phi = phi(p),  n = phi;

  for (ll res = 2; res <= p; res++) {
    bool ok = true;
    for (auto &f : facs)
      ok &= powMod(res, phi / f.first, p) != 1;
    if (ok)  return res;
  }
  return -1;
}
```

4.8 Diskreter Logarithmus

```cpp
// Bestimmt Lösung x für a^x=b mod m.
ll solve (ll a, ll b, ll m) { // Laufzeit: O(sqrt(m)*log(m))
  ll n = (ll)sqrt((double)m) + 1;
  map<ll,ll> vals;
  for (int i = n; i >= 1; i--) vals[powMod(a, i * n, m)] = i;
  for (int i = 0; i <= n; i++) {
    ll cur = (powMod(a, i, m) * b) % m;
    if (vals.count(cur)) {
      ll ans = vals[cur] * n - i;
      if (ans < m) return ans;
  }}
  return -1;
}
```

4.9 Binomialkoeffizienten

```cpp
// 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;
}
```

4.10 LGS über $\mathbb{F}_p$

```
// 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);
}}
```

4.11 LGS über $\mathbb{R}$

```
// 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;
}
```

4.12 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
```

4.13 Numerisch Integrieren, Simpsonregel

```
double f(double x) { return x; }

double simps(double a, double b) {
  return (f(a) + 4.0 * f((a + b) / 2.0) + f(b)) * (b - a) / 6.0;
}

double integrate(double a, double b) {
  double m = (a + b) / 2.0;
  double l = simps(a, m), r = simps(m, b), tot = simps(a, b);
  if (abs(l + r - tot) < EPSILON) return tot;
  return integrate(a, m) + integrate(m, b);
}
```

4.14 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);
  }
};
```

4.15 Longest Increasing Subsequence

```
vector<int> longestIncreasingSubsequence(vector<int> &seq) {
  int n = seq.size(), lisLength = 0, lisEnd = 0;
  vector<int> L(n), L_id(n), parents(n);
  for (int i = 0; i < n; i++) {
    int pos =
      lower_bound(L.begin(), L.begin() + lisLength, seq[i]) - L.begin();
    L[pos] = seq[i];
    L_id[pos] = i;
    parents[i] = pos ? L_id[pos - 1] : -1;
    if (pos + 1 > lisLength) {
      lisLength = pos + 1;
      lisEnd = i;
  }}
  // Ab hier Rekonstruktion der Sequenz.
  vector<int> result(lisLength);
  int pos = lisLength - 1, x = lisEnd;
  while (parents[x] >= 0) {
    result[pos--] = x;
    x = parents[x];
  }
  result[0] = x;
  return result; // Liste mit Indizes einer LIS.
}
```

4.16 Inversionszahl und Mergesort

```
// Laufzeit: O(n*log(n))
ll merge(vector<ll> &v, vector<ll> &left, vector<ll> &right) {
  int a = 0, b = 0, i = 0;
  ll inv = 0;
  while (a < (int)left.size() && b < (int)right.size()) {
    if (left[a] < right[b]) v[i++] = left[a++];
    else {
      inv += left.size() - a;
      v[i++] = right[b++];
    }
  }
  while (a < (int)left.size()) v[i++] = left[a++];
  while (b < (int)right.size()) v[i++] = right[b++];
  return inv;
```

```
}

ll mergeSort(vector<ll> &v) { // Sortiert v und gibt Inversionszahl zurück.
  int n = v.size();
  vector<ll> left(n / 2), right((n + 1) / 2);
  for (int i = 0; i < n / 2; i++) left[i] = v[i];
  for (int i = n / 2; i < n; i++) right[i - n / 2] = v[i];

  ll result = 0;
  if (left.size() > 1) result += mergeSort(left);
  if (right.size() > 1) result += mergeSort(right);
  return result + merge(v, left, right);
}
```

4.17 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 $X_1, \ldots, X_k$ hat, dann ist die Grundy-Zahl des Gesamtzustandes $g(X_1) \oplus \ldots \oplus g(X_k)$.

4.18 Legendre-Symbol

Sei $p \geq 3$ eine Primzahl, $a \in \mathbb{Z}$:

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{falls } p \mid a \\ 1 & \text{falls } \exists x \in \mathbb{Z} \backslash p\mathbb{Z} : a \equiv x^2 \mod p \\ -1 & \text{sonst} \end{cases}$$

$$\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}} = \begin{cases} 1 & \text{falls } p \equiv 1 \mod 4 \\ -1 & \text{falls } p \equiv 3 \mod 4 \end{cases}$$

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1 & \text{falls } p \equiv \pm 1 \mod 8 \\ -1 & \text{falls } p \equiv \pm 3 \mod 8 \end{cases}$$

$$\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2} \cdot \frac{q-1}{2}}$$

```
int legendre(ll a, ll p) {
  a %= p;
  if (a == 0) return 0;
  if (a == 1 || p == 2) return 1;
  if (a == 2) return (((p * p - 1) / 8) & 1) ? -1 : 1;
  if (isPrime(a)) {
    return legendre(p, a) * ((((p - 1) * (a - 1) / 4) & 1) ? -1 : 1);
  } else {
    map<ll, int> facts;
    factor(a, facts);
    int res = 1;
    for (auto f : facts)
      if (f.second & 1)
        res *= legendre(f.first, p);
    return res;
  }
}
```

4.19 Kombinatorik

Berühmte Zahlen	
Fibonacci	$f(0) = 0 \quad f(1) = 1 \quad f(n+2) = f(n+1) + f(n)$
Catalan	$C_0 = 1 \qquad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1}\binom{2n}{n} = \frac{2(2n-1)}{n+1} \cdot C_{n-1}$
Euler I	$\left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1 \qquad \left\langle {n \atop k} \right\rangle = (k+1)\left\langle {n-1 \atop k} \right\rangle + (n-k)\left\langle {n-1 \atop k-1} \right\rangle$
Euler II	$\left\langle\!\left\langle {n \atop 0} \right\rangle\!\right\rangle = 1 \quad \left\langle\!\left\langle {n \atop n} \right\rangle\!\right\rangle = 0 \quad \left\langle\!\left\langle {n \atop k} \right\rangle\!\right\rangle = (k+1)\left\langle\!\left\langle {n-1 \atop k} \right\rangle\!\right\rangle + (2n-k-1)\left\langle\!\left\langle {n-1 \atop k-1} \right\rangle\!\right\rangle$
Stirling I	$\left[{0 \atop 0}\right] = 1 \qquad \left[{n \atop 0}\right] = \left[{0 \atop n}\right] = 0 \qquad \left[{n \atop k}\right] = \left[{n-1 \atop k-1}\right] + (n-1)\left[{n-1 \atop k}\right]$
Stirling II	$\left\{{n \atop 1}\right\} = \left\{{n \atop n}\right\} = 1 \qquad \left\{{n \atop k}\right\} = k\left\{{n-1 \atop k}\right\} + \left\{{n-1 \atop k-1}\right\}$
Bell	$B_1 = 1 \qquad B_n = \sum_{k=0}^{n-1} B_k \binom{n-1}{k} = \sum_{k=0}^{n} \left\{{n \atop k}\right\}$
Partitions	$f(0,0) = 1 \quad f(n,k) = 0$ für $k > n$ oder $n \leq 0$ oder $k \leq 0$ $f(n,k) = f(n-k,k) + f(n-1,k-1)$

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.

Catalan-Zahlen

- 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.
- 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 $n \times n$-Gitter, die nicht die Diagonale kreuzen.

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

Euler-Zahlen 2. Ordnung Die Anzahl der Permutationen von $\{1, 1, \ldots, n, n\}$ mit genau k Anstiegen.

Stirling-Zahlen 1. Ordnung Die Anzahl der Permutationen von $\{1, \ldots, n\}$ mit genau k Zyklen. 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.

Stirling-Zahlen 2. Ordnung Die Anzahl der Möglichkeiten n Elemente in k nichtleere Teilmengen zu zerlegen. 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.

Bell-Zahlen Anzahl der Partitionen von $\{1, \ldots, n\}$. Wie Stirling-Zahlen 2. Ordnung ohne Limit durch k.

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

Binomialkoeffizienten

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$	$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$	$\sum_{k=0}^{n} \binom{r+k}{k} = \binom{r+n+1}{n}$
$\sum_{k=0}^{n} \binom{n}{k} = 2^n$	$\binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}$	$\binom{n}{k} = (-1)^k \binom{k-n-1}{k}$
$\binom{n}{k} = \binom{n}{n-k}$	$\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1}$	$\sum_{i=0}^{n} \binom{n}{i}^2 = \binom{2n}{n}$
$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1}$	$\sum_{k=0}^{n} \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}$	$\sum_{i=1}^{n} \binom{n}{i} F_i = F_{2n}$ $\quad F_n = n$-th Fib.

The Twelvefold Way (verteile n Bälle auf k Boxen)

Bälle / Boxen	identisch / identisch	unterscheidbar / identisch	identisch / unterscheidbar	unterscheidbar / unterscheidbar
-	$p_k(n)$	$\sum_{i=0}^{k} \left\{ n \atop i \right\}$	$\binom{n+k-1}{k-1}$	k^n
size ≥ 1	$p(n,k)$	$\left\{ n \atop k \right\}$	$\binom{n-1}{k-1}$	$k! \left\{ n \atop k \right\}$
size ≤ 1	$[n \leq k]$	$[n \leq k]$	$\binom{k}{n}$	$n! \binom{k}{n}$

$p_k(n)$: #Anzahl der Partitionen von n in $\leq k$ positive Summanden.
$p(n,k)$: #Anzahl der Partitionen von n in genau k positive Summanden.
[Bedingung]: `return Bedingung ? 1 : 0;`

Platonische Körper

Übersicht	Seiten	Ecken	Kanten	dual zu
Tetraeder	4	4	6	Tetraeder
Würfel/Hexaeder	6	8	12	Oktaeder
Oktaeder	8	6	12	Würfel/Hexaeder
Dodekaeder	12	20	30	Ikosaeder
Ikosaeder	20	12	30	Dodekaeder

Färbungen mit maximal n Farben (bis auf Isomorphie)

Ecken vom Oktaeder/Seiten vom Würfel	$(n^6 + 3n^4 + 12n^3 + 8n^2)/24$
Ecken vom Würfel/Seiten vom Oktaeder	$(n^8 + 17n^4 + 6n^2)/24$
Kanten vom Würfel/Oktaeder	$(n^{12} + 6n^7 + 3n^6 + 8n^4 + 6n^3)/24$
Ecken/Seiten vom Tetraeder	$(n^4 + 11n^2)/12$
Kanten vom Tetraeder	$(n^6 + 3n^4 + 8n^2)/12$
Ecken vom Ikosaeder/Seiten vom Dodekaeder	$(n^{12} + 15n^6 + 44n^4)/60$
Ecken vom Dodekaeder/Seiten vom Ikosaeder	$(n^{20} + 15n^{10} + 20n^8 + 24n^4)/60$
Kanten vom Dodekaeder/Ikosaeder (evtl. falsch)	$(n^{30} + 15n^{16} + 20n^{10} + 24n^6)/60$

Wahrscheinlichkeitstheorie (A, B Ereignisse und X, Y Variablen)

$E(X+Y) = E(X) + E(Y)$	$E(\alpha X) = \alpha E(X)$
X, Y unabh. $\Leftrightarrow E(XY) = E(X) \cdot E(Y)$	$\Pr[A \mid B] = \frac{\Pr[A \land B]}{\Pr[B]}$
$\Pr[A \lor B] = \Pr[A] + \Pr[B] - \Pr[A \land B]$	$\Pr[A \land B] = \Pr[A] \cdot \Pr[B]$

Bertrand's Ballot Theorem (Kandidaten A und B, $k \in \mathbb{N}$)

$\#A > k\#B$	$Pr = \frac{a-kb}{a+b}$	$\#B - \#A \leq k$	$Pr = 1 - \frac{a!b!}{(a+k+1)!(b-k-1)!}$
$\#A \geq k\#B$	$Pr = \frac{a+1-kb}{a+1}$	$\#A \geq \#B + k$	$Num = \frac{a-k+1-b}{a-k+1}\binom{a+b-k}{b}$

Nim-Spiele (❶ letzter gewinnt (normal), ❷ letzter verliert)

Beschreibung	Strategie
$M = [pile_i]$ $[x] := \{1, \dots, x\}$	$SG = \oplus_{i=1}^{n} pile_i$ ❶ Nimm von einem Stapel, sodass SG 0 wird. ❷ Genauso. Außer: Bleiben nur noch Stapel der Größe 1, erzeuge ungerade Anzahl solcher Stapel.
$M = \{a^m \mid m \geq 0\}$	a ungerade: $SG_n = n\%2$ a gerade: $SG_n = 2$, falls $n \equiv a \mod (a+1)$ $SG_n = n\%(a+1)\%2$, sonst.
$M_{①} = \left[\frac{pile_i}{2}\right]$ $M_{②} = \left\{\left[\frac{pile_i}{2}\right], pile_i\right\}$	① $SG_{2n} = n$, $SG_{2n+1} = SG_n$ ② $SG_0 = 0$, $SG_n = [\log_2 n] + 1$
$M_{①}$ = Teiler von $pile_i$ $M_{②}$ = echte Teiler von $pile_i$	① $SG_0 = 0$, $SG_n = SG_{②,n} + 1$ ② $ST_1 = 0$, SG_n = #Nullen am Ende von n_{bin}
$M_{①} = [k]$ $M_{②} = S$, (S endlich) $M_{③} = S \cup \{pile_i\}$	$SG_{①,n} = n \mod (k+1)$ ❶ Niederlage bei $SG = 0$ ❷ Niederlage bei $SG = 1$ $SG_{③,n} = SG_{②,n} + 1$
Für jedes endliche M ist SG eines Stapels irgendwann periodisch.	
Moore's Nim: Beliebige Zahl von maximal k Stapeln.	❶ Schreibe $pile_i$ binär. Addiere ohne Übertrag zur Basis $k+1$. Niederlage, falls Ergebnis gleich 0. ❷ Wenn alle Stapel 1 sind: Niederlage, wenn $n \equiv 1 \mod (k+1)$. Sonst wie in ❶.
Staircase Nim: n Stapel in einer Reihe. Beliebige Zahl von Stapel i nach Stapel $i-1$.	Niederlage, wenn Nim der ungeraden Spiele verloren ist: $\oplus_{i=0}^{(n-1)/2} pile_{2i+1} = 0$
Lasker's Nim: Zwei mögliche Züge: 1) Nehme beliebige Zahl. 2) Teile Stapel in zwei Stapel (ohne Entnahme).	$SG_n = n$, falls $n \equiv 1,2 \mod 4$ $SG_n = n+1$, falls $n \equiv 3 \mod 4$ $SG_n = n-1$, falls $n \equiv 0 \mod 4$
Kayles' Nim: Zwei mögliche Züge: 1) Nehme beliebige Zahl. 2) Teile Stapel in zwei Stapel (mit Entnahme).	Berechne SG_n für kleine n rekursiv. $n \in [72, 83]$: 4, 1, 2, 8, 1, 4, 7, 2, 1, 8, 2, 7 Periode ab $n = 72$ der Länge 12.

Reihen

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$	$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(n+2)}{6}$	$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$
$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1} \quad c \neq 1$	$\sum_{i=0}^{\infty} c^i = \frac{1}{1-c} \quad \lvert c\rvert < 1$	$\sum_{i=1}^{\infty} c^i = \frac{c}{1-c} \quad \lvert c\rvert < 1$
$\sum_{i=0}^{n} ic^i = \frac{nc^{n+2}-(n+1)c^{n+1}+c}{(c-1)^2} \quad c \neq 1$		$\sum_{i=0}^{\infty} ic^i = \frac{c}{(1-c)^2} \quad \lvert c\rvert < 1$
$H_n = \sum_{i=1}^{n} \frac{1}{i}$	$\sum_{i=1}^{n} iH_i = \frac{n(n+1)}{2}H_n - \frac{n(n-1)}{4}$	
$\sum_{i=1}^{n} H_i = (n+1)H_n - n$	$\sum_{i=1}^{n} \binom{i}{m} H_i = \binom{n+1}{m+1}\left(H_{n+1} - \frac{1}{m+1}\right)$	

Verschiedenes

Türme von Hanoi, minimale Schirttzahl:	$T_n = 2^n - 1$
#Regionen zwischen n Gearden	$\frac{n(n+1)}{2} + 1$
#abgeschlossene Regionen zwischen n Geraden	$\frac{n^2-3n+2}{2}$
#markierte, gewurzelte Bäume	n^{n-1}
#markierte, nicht gewurzelte Bäume	n^{n-2}
#Wälder mit k gewurzelten Bäumen	$\frac{k}{n}\binom{n}{k}n^{n-k}$
Dearangements	$!n = (n-1)(!(n-1)+!(n-2))$ $\lim_{n\to\infty} \frac{!n}{n!} = \frac{1}{e}$

4.20 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;
  }

  // Gibt die Zahl aus.
  void print() {
    if (digits.size() == 0) { printf("0"); return; }
    if (sign == MINUS) printf("-");
    printf("%lld", digits[digits.size() - 1]);
    for (int i = digits.size() - 2; i >= 0; i--) {
      printf("%09lld", digits[i]); // Anpassen, wenn andere Basis gewählt wird.
  }}
};

// Kleiner-oder-gleich-Vergleich.
bool operator<=(bigint &a, bigint &b) {
  if (a.digits.size() == b.digits.size()) {
    int idx = a.digits.size() - 1;
    while (idx >= 0) {
      if (a.digits[idx] < b.digits[idx]) return true;
      else if (a.digits[idx] > b.digits[idx]) return false;
      idx--;
    }
    return true;
  }
  return a.digits.size() < b.digits.size();
}

// Kleiner-Vergeleich.
bool operator<(bigint &a, bigint &b) {
  if (a.digits.size() == b.digits.size()) {
    int idx = a.digits.size() - 1;
    while (idx >= 0) {
      if (a.digits[idx] < b.digits[idx]) return true;
      else if (a.digits[idx] > b.digits[idx]) return false;
      idx--;
    }
    return false;
  }
  return a.digits.size() < b.digits.size();
}

void sub(bigint *a, bigint *b, bigint *c);

// a + b = c. a, b, c dürfen gleich sein.
void add(bigint *a, bigint *b, bigint *c) {
  if (a->sign == b->sign) c->sign = a->sign;
  else {
    if (a->sign == MINUS) {
      a->sign ^= 1;
      sub(b, a, c);
      a->sign ^= 1;
    } else {
      b->sign ^= 1;
      sub(a, b, c);
      b->sign ^= 1;
    }
    return;
  }

  c->digits.resize(max(a->digits.size(), b->digits.size()));
  ll carry = 0;
  int i = 0;
  for (; i < (int)min(a->digits.size(), b->digits.size()); i++) {
    ll sum = carry + a->digits[i] + b->digits[i];
    c->digits[i] = sum % BASE;
    carry = sum / BASE;
  }
  if (i < (int)a->digits.size()) {
    for (; i< (int)a->digits.size(); i++) {
      ll sum = carry + a->digits[i];
      c->digits[i] = sum % BASE;
      carry = sum / BASE;
    }
  } else {
    for (; i< (int)b->digits.size(); i++) {
      ll sum = carry + b->digits[i];
      c->digits[i] = sum % BASE;
      carry = sum / BASE;
  }}
  if (carry) c->digits.push_back(carry);
}

// a - b = c. c darf a oder b sein. a und b müssen verschieden sein.
```

```
void sub(bigint *a, bigint *b, bigint *c) {
  if (a->sign == MINUS || b->sign == MINUS) {
    b->sign ^= 1;
    add(a, b, c);
    b->sign ^= 1;
    return;
  }

  if (a < b) {
    sub(b, a, c);
    c->sign = MINUS;
    c->trim();
    return;
  }

  c->digits.resize(a->digits.size());
  ll borrow = 0;
  int i = 0;
  for (; i < (int)b->digits.size(); i++) {
    ll diff = a->digits[i] - borrow - b->digits[i];
    if (a->digits[i] > 0) borrow = 0;
    if (diff < 0) {
      diff += BASE;
      borrow = 1;
    }
    c->digits[i] = diff % BASE;
  }
  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();
}

// Ziffernmultiplikation 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: aho.push_back(vertex());
// 2) Mit addString(0, pattern, idx); Patterns hinzufügen.
// 3) finishAutomaton(0) aufrufen.
// 4) Mit state = go(state, 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 {
  int next[ALPHABET_SIZE], failure;
  int character;
  vector<int> patterns; // Indizes der Patterns, die hier enden.
  vertex() { for (int i = 0; i < ALPHABET_SIZE; i++) next[i] = -1; }
};
vector<vertex> aho;

void addString(int v, vector<int> &pattern, int patternIdx) {
  for (int i = 0; i < (int)pattern.size(); i++) {
    if (aho[v].next[pattern[i]] == -1) {
      aho[v].next[pattern[i]] = aho.size();
      aho.push_back(vertex());
      aho.back().character = pattern[i];
    }
    v = aho[v].next[pattern[i]];
  }
  aho[v].patterns.push_back(patternIdx);
}

void finishAutomaton(int v) {
  for (int i = 0; i < ALPHABET_SIZE; i++)
    if (aho[v].next[i] == -1) aho[v].next[i] = v;

  queue<int> q;
  for (int i = 0; i < ALPHABET_SIZE; i++) {
    if (aho[v].next[i] != v) {
      aho[aho[v].next[i]].failure = v;
      q.push(aho[v].next[i]);
  }}
  while (!q.empty()) {
    int r = q.front(); q.pop();
    for (int i = 0; i < ALPHABET_SIZE; i++) {
      if (aho[r].next[i] != -1) {
        q.push(aho[r].next[i]);
        int f = aho[r].failure;
        while (aho[f].next[i] == -1) f = aho[f].failure;
        aho[aho[r].next[i]].failure = aho[f].next[i];
        for (int j = 0; j < (int)aho[aho[f].next[i]].patterns.size(); j++) {
          aho[aho[r].next[i]].patterns.push_back(
              aho[aho[f].next[i]].patterns[j]);
}}}}}

int go(int v, int c) {
  if (aho[v].next[c] != -1) return aho[v].next[c];
  else return go(aho[v].failure, c);
}
```

5.3 Trie

```
// Zahlenwerte müssen bei 0 beginnen und zusammenhängend sein.
struct node {
  int children[ALPHABET_SIZE], c; // c = #Wörter, die hier enden.
  node () {
    idx = -1;
```

```
    for (int i = 0; i < ALPHABET_SIZE; i++) children[i] = -1;
  }
};
vector<node> trie; // Anlegen mit trie.push_back(node());

void insert(int vert, vector<int> &txt, int s) { // Laufzeit: O(|txt|)
  if (s == (int)txt.size()) { trie[vert].c++; return; }
  if (trie[vert].children[txt[s]] == -1) {
    trie[vert].children[txt[s]] = trie.size();
    trie.push_back(node());
  }
  insert(trie[vert].children[txt[s]], txt, s + 1);
}

int contains(int vert, vector<int> &txt, int s) { // Laufzeit: O(|txt|)
  if (s == (int)txt.size()) return trie[vert].c;
  if (trie[vert].children[txt[s]] != -1) {
    return contains(trie[vert].children[txt[s]], txt, s + 1);
  } else return 0;
}
```

5.4 Suffix-Baum

```
// Baut Suffixbaum online auf. Laufzeit: O(n)
// Einmal initSuffixTree() aufrufen und dann extend für jeden Buchstaben.
// '\0'-Zeichen (oder ähnliches) an den Text anhängen!
string s;
int root, lastIdx, needsSuffix, pos, remainder, curVert, curEdge, curLen;
struct Vert {
  int start, end, suffix; // Kante [start,end)
  map<char, int> next;
  int len() { return min(end, pos + 1) - start; }
};
vector<Vert> tree;

int newVert(int start, int end) {
  Vert v;
  v.start = start;
  v.end = end;
  v.suffix = 0;
  tree.push_back(v);
  return ++lastIdx;
}

void addSuffixLink(int vert) {
  if (needsSuffix) tree[needsSuffix].suffix = vert;
  needsSuffix = vert;
}

bool fullImplicitEdge(int vert) {
  if (curLen >= tree[vert].len()) {
    curEdge += tree[vert].len();
    curLen -= tree[vert].len();
    curVert = vert;
    return true;
  }
  return false;
}

void initSuffixTree() {
  needsSuffix = remainder = curEdge = curLen = 0;
  lastIdx = pos = -1;
  root = curVert = newVert(-1, -1);
}

void extend() {
  pos++;
  needsSuffix = 0;
  remainder++;
  while (remainder) {
    if (curLen == 0) curEdge = pos;
    if (!tree[curVert].next.count(s[curEdge])) {
      int leaf = newVert(pos, s.size());
      tree[curVert].next[s[curEdge]] = leaf;
      tree[curVert].next[s[curEdge]] = leaf;
      addSuffixLink(curVert);
    } else {
      int nxt = tree[curVert].next[s[curEdge]];
      if (fullImplicitEdge(nxt)) continue;
      if (s[tree[nxt].start + curLen] == s[pos]) {
        curLen++;
        addSuffixLink(curVert);
        break;
      }
      int split = newVert(tree[nxt].start, tree[nxt].start + curLen);
      tree[curVert].next[s[curEdge]] = split;
      int leaf = newVert(pos, s.size());
      tree[split].next[s[pos]] = leaf;
      tree[nxt].start += curLen;
      tree[split].next[s[tree[nxt].start]] = nxt;
      addSuffixLink(split);
    }
    remainder--;
    if (curVert == root && curLen) {
      curLen--;
      curEdge = pos - remainder + 1;
    } else {
      curVert = tree[curVert].suffix ? tree[curVert].suffix : root;
    }
  }
}
```

5.5 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;
  }
};
```

5.6 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) {
    c -= 'a'; // Werte von c müssen bei 0 beginnen.
    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);
  }

  // Berechnet die Terminale des Automaten.
  vector<int> calculateTerminals() {
    vector<int> terminals;
    int pos = last;
    while (pos != -1) {
      terminals.push_back(pos);
      pos = states[pos].link;
    }
    return terminals;
  }
};
```

- **Ist `w` Substring von `s`?** Baue Automaten für `s` und wende ihn auf `w` an. Wenn alle Übergänge vorhanden sind, ist `w` Substring von `s`.
- **Ist `w` Suffix von `s`?** Wie oben. Überprüfe am Ende, ob aktueller Zustand ein

Terminal ist.

- **Anzahl verschiedener Substrings.** Jeder Pfad im Automaten entspricht einem Substring. Für einen Knoten ist die Anzahl der ausgehenden Pfade gleich der Summe über die Anzahlen der Kindknoten plus 1. Der letzte Summand ist der Pfad, der in diesem Knoten endet.
- **Wie oft taucht `w` in `s` auf?** Sei `p` der Zustand nach Abarbeitung von `w`. Lösung ist Anzahl der Pfade, die in `p` starten und in einem Terminal enden. Diese Zahl lässt sich wie oben rekursiv berechnen. Bei jedem Knoten darf nur dann plus 1 gerechnet werden, wenn es ein Terminal ist.

5.7 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;
}
```

5.8 Rolling Hash

```
ll q = 31; // Größer als Alphabetgröße. q=31,53,311
struct Hasher {
  string s;
  ll mod;
  vector<ll> power, pref;
  Hasher(const string& s, ll mod) : s(s), mod(mod) {
    power.push_back(1);
    for (int i = 1; i < (int)s.size(); i++)
      power.push_back(power.back() * q % mod);
    pref.push_back(0);
    for (int i = 0; i < (int)s.size(); i++)
      pref.push_back((pref.back() * q % mod + s[i]) % mod);
  }

  // Berechnet hash(s[l..r]). l,r inklusive.
  ll hash(int l, int r) {
    return (pref[r+1] - power[r-l+1] * pref[l] % mod + mod) % mod;
  }
};
```

5.9 MANACHER's Algorithm, Longest Palindrome

```
char input[MAX_N];
char s[2 * MAX_N + 1];
int longest[2 * MAX_N + 1];

void setDots() {
  s[0] = '.';
  int j = 1;
  for (int i = 0; i < (int)strlen(input); i++) {
    s[j++] = input[i];
    s[j++] = '.';
  }
  s[j] = '\0';
}

void manacher() {
  int center = 0, last = 0, n = strlen(s);
  memset(longest, 0, sizeof(longest));

  for (int i = 1; i < n - 1; i++) {
    int i2 = 2 * center - i;
    longest[i] = (last > i) ? min(last - i, longest[i2]) : 0;
    while (i + longest[i] + 1 < n && i - longest[i] - 1 >= 0 &&
        s[i + longest[i] + 1] == s[i - longest[i] - 1]) longest[i]++;
    if (i + longest[i] > last) {
      center = i;
      last = i + longest[i];
    }
  }
  for (int i = 0; i < n; i++) longest[i] = 2 * longest[i] + 1;
}
```

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 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;
}
```

7.2 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
// Iteriert über alle Teilmengen einer Bitmaske (außer der leeren Menge).
for (int subset = bitmask; subset > 0; subset = (subset - 1) & bitmask)
```

7.3 Fast IO

```
void fastscan(int* number) {
  bool negative = false;
  register int c;
  *number = 0;
  c = getchar();
  while(c != '-' && (c < '0' || c > '9')) c = getchar();
  if (c == '-') negative = true, c = getchar();
  for (; c > 47 && c < 58; c = getchar()) *number = *number * 10 + c -
      48;
  if (negative) *number *= -1;
}

void printPositive(int n) {
  if (n == 0) return;
  print(n / 10);
  putchar(n % 10 + '0');
}

void fastprint(int n) {
  if(n == 0) { putchar('0'); return; }
  if (n < 0) {
    putchar('-');
    print(-n);
  } else print(n);
}
```

7.4 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/Float. Muss zum Einlesen/Ausgeben in einen int oder long long gecastet werden.
__int128, __float128
```

7.5 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 $0, 1, \dots, n-1$. 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: $F(1,k) = 0$.

```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 $1, \dots, n$ nummeriert sind, im zweiten Fall von $0, \dots, n-1$!

7.6 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.
- **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 9)
- **Satz von Pick:** Sei A der Flächeninhalt eines einfachen Gitterpolygons, I die Anzahl der Gitterpunkte im Inneren und R die Anzahl der Gitterpunkte auf dem Rand. Es gilt:
  $$A = I + \frac{R}{2} - 1$$
- **Lemma von Burnside:** Sei G eine endliche Gruppe, die auf der Menge X operiert. Für jedes $g \in G$ sei X^g die Menge der Fixpunkte bei Operation durch g, also $X^g = \{x \in X \mid g \bullet x = x\}$. Dann gilt für die Anzahl der Bahnen $[X/G]$ der Operation:
  $$[X/G] = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

- **Verteilung von Primzahlen:** Für alle $n \in \mathbb{N}$ gilt: Ex existiert eine Primzahl p mit $n \leq p \leq 2n$.
- **Satz von Kirchhoff:** Sei G ein zusammenhängender, ungerichteter Graph evtl. mit Mehrfachkanten. Sei A die Adjazenzmatrix von G. Dabei ist a_{ij} die Anzahl der Kanten zwischen Knoten i und j. Sei B eine Diagonalmatrix, b_{ii} sei der Grad von Knoten i. Definiere $R = B - A$. Alle Kofaktoren von R sind gleich und die Anzahl der Spannbäume von G.
  Entferne letzte Zeile und Spalte und berechne Betrag der Determinante.
- **Dilworth's-Theorem:** Sei S eine Menge und $\leq$ eine partielle Ordnung (S ist ein Poset). Eine *Kette* ist eine Teilmenge $\{x_1, \ldots, x_n\}$ mit $x_1 \leq \ldots \leq x_n$. Eine *Partition* ist eine Menge von Ketten, sodass jedes $s \in S$ in genau einer Kette ist. Eine *Antikette* ist eine Menge von Elementen, die paarweise nicht vergleichbar sind.
  Es gilt: Die Größe der längsten Antikette gleicht der Größe der kleinsten Partition. $\Rightarrow$ *Weite* des Poset.
  Berechnung: Maximales Matching in bipartitem Graphen. Dupliziere jedes $s \in S$ in u_s und v_s. Falls $x \leq y$, füge Kante $u_x \to v_y$ hinzu. Wenn Matching zu langsam ist, versuche Struktur des Posets auszunutzen und evtl. anders eine maximale Anitkette zu finden.

7.7 Tipps & Tricks

- Run Tim Error:
  - Stack Overflow? Evtl. rekurisve Tiefensuche auf langem Pfad?
  - Array-Grenzen überprüfen. Indizierung bei 0 oder bei 1 beginnen?
  - Abbruchbedingung bei Rekursion?
  - Evtl. Memory Limit Exceeded?
- Gleitkommazahlen:
  - `NaN`? Evtl. ungültige Werte für mathematische Funktionen, z.B. `acos(1.00000000000001)`?
  - Flasches Runden bei negativen Zahlen? Abschneiden $\neq$ Abrunden!
  - Output in wissenschaftlicher Notation (`1e-25`)?
  - Kann `-0.000` ausgegeben werden?
- Wrong Answer:
  - Lies Aufgabe erneut. Sorgfältig!
  - Mehrere Testfälle in einer Datei? Probiere gleichen Testcase mehrfach hintereinander.
  - Integer Overflow? Teste maximale Eingabegrößen und mache Überschlagsrechnung.
  - Einabegrößen überprüfen. Sonderfälle ausprobieren.
    - $n = 0, n = -1, n = 1, n = 2^{31} - 1, n = -2^{31}$
    - n gerade/ungerade
    - Graph ist leer/enthält nur einen Knoten.
    - Liste ist leer/enthält nur ein Element.
    - Graph ist Multigraph (enthält Schleifen/Mehrfachkanten).
    - Sind Kanten gerichtet/ungerichtet?
    - Polygon ist konkav/selbstschneidend.
  - Bei DP/Rekursion: Stimmt Basisfall?
  - Unsicher bei benutzten STL-Funktionen?