

26. November 2017

## 1 Datenstrukturen

### 1.1 Union-Find

```

1 // Laufzeit: O(n*alpha(n))
2 // "height" ist obere Schranke für die Höhe der Bäume. Sobald
3 // Pfadkompression angewendet wurde, ist die genaue Höhe nicht mehr
4 // effizient berechenbar.
5 vector<int> parent; // Initialisiere mit Index im Array.
6 vector<int> height; // Initialisiere mit 0.
7
8 int findSet(int n) { // Pfadkompression
9     if (parent[n] != n) parent[n] = findSet(parent[n]);
10    return parent[n];
11 }
12
13 void linkSets(int a, int b) { // Union by rank.
14     if (height[a] < height[b]) parent[a] = b;
15     else if (height[b] < height[a]) parent[b] = a;
16     else {
17         parent[a] = b;
18         height[b]++;
19     }
20
21 void unionSets(int a, int b) { // Diese Funktion aufrufen.
22     if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
23 }

```

### 1.2 Segmentbaum

```

1 // Laufzeit: init: O(n), query: O(log n), update: O(log n)
2 // Berechnet das Maximum im Array.
3 int a[MAX_N], m[4 * MAX_N];
4
5 int query(int x, int y, int k = 0, int X = 0, int Y = MAX_N - 1) {
6     if (x <= X && Y <= y) return m[k];
7     if (y < X || Y < x) return -INF; // Ein "neutrales" Element.
8     int M = (X + Y) / 2;
9     return max(query(x, y, 2*k+1, X, M), query(x, y, 2*k+2, M+1, Y));
10 }
11
12 void update(int i, int v, int k = 0, int X = 0, int Y = MAX_N - 1) {
13     if (i < X || Y < i) return;
14     if (X == Y) { m[k] = v; a[i] = v; return; }
15     int M = (X + Y) / 2;
16     update(i, v, 2 * k + 1, X, M);
17     update(i, v, 2 * k + 2, M + 1, Y);
18     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
19 }
20
21 void init(int k = 0, int X = 0, int Y = MAX_N - 1) {
22     if (X == Y) { m[k] = a[X]; return; }

```

```

23     int M = (X + Y) / 2;
24     init(2 * k + 1, X, M);
25     init(2 * k + 2, M + 1, Y);
26     m[k] = max(m[2 * k + 1], m[2 * k + 2]);
27 }

```

### 1.3 2D-Segmentbaum

```

1 // 1-indiziert. Array t: [4*n][4*m]. Nur die _x-Varianten aufrufen.
2 // Laufzeit: build: O(n*m), update, sum: O(log(n)*log(m))
3 void build_y(int vx, int lx, int rx, int vy, int ly, int ry) {
4     if (ly == ry) {
5         if (lx == rx) t[vx][vy] = a[lx][ly];
6         else t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
7     } else {
8         int my = (ly + ry) / 2;
9         build_y(vx, lx, rx, vy*2, ly, my);
10        build_y(vx, lx, rx, vy*2+1, my+1, ry);
11        t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
12    }
13
14 void build_x(int vx = 1, int lx = 0, int rx = N-1) {
15     if (lx != rx) {
16         int mx = (lx + rx) / 2;
17         build_x(vx*2, lx, mx);
18         build_x(vx*2+1, mx+1, rx);
19     }
20     build_y(vx, lx, rx, 1, 0, m-1);
21 }
22
23 int sum_y(int vx, int vy, int tly, int try_, int ly, int ry) {
24     if (ly > ry) return 0;
25     if (ly == tly && try_ == ry) return t[vx][vy];
26     int tmy = (tly + try_) / 2;
27     return sum_y(vx, vy*2, tly, tmy, ly, min(ry, tmy))
28         + sum_y(vx, vy*2+1, tmy+1, try_, max(ly, tmy+1), ry);
29 }
30
31 int sum_x(int vx=1, int tlx=0, int trx=n-1, int lx, int rx, int ly, int ry) {
32     if (lx > rx) return 0;
33     if (lx == tlx && trx == rx) return sum_y(vx, 1, 0, m-1, ly, ry);
34     int tmx = (tlx + trx) / 2;
35     return sum_x(vx*2, tlx, tmx, lx, min(rx, tmx), ly, ry)
36         + sum_x(vx*2+1, tmx+1, trx, max(lx, tmx+1), rx, ly, ry);
37 }
38
39 void update_y(int vx, int lx, int rx, int vy, int ly, int ry,
40     int x, int y, int new_val) {
41     if (ly == ry) {
42         if (lx == rx) t[vx][vy] = new_val;
43         else t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];

```

```

44 } else {
45     int my = (ly + ry) / 2;
46     if (y <= my) update_y(vx, lx, rx, vy*2, ly, my, x, y, new_val);
47     else update_y(vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
48     t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
49 }
50
51 void update_x(int vx=1, int lx=0, int rx=n-1, int x, int y, int new_val) {
52     if (lx != rx) {
53         int mx = (lx + rx) / 2;
54         if (x <= mx) update_x(vx*2, lx, mx, x, y, new_val);
55         else update_x(vx*2+1, mx+1, rx, x, y, new_val);
56     }
57     update_y(vx, lx, rx, 1, 0, m-1, x, y, new_val);
58 }

```

## 1.4 Fenwick Tree

```

1 vector<int> FT; // Fenwick-Tree
2 int n;
3
4 // Addiert val zum Element an Index i. O(log(n)).
5 void updateFT(int i, int val) {
6     i++; while(i <= n) { FT[i] += val; i += (i & (-i)); }
7 }
8
9 // Baut Baum auf. O(n*log(n)).
10 void buildFenwickTree(vector<int>& a) {
11     n = a.size();
12     FT.assign(n+1,0);
13     for(int i = 0; i < n; i++) updateFT(i, a[i]);
14 }
15
16 // Präfix-Summe über das Intervall [0..i]. O(log(n)).
17 int prefix_sum(int i) {
18     int sum = 0; i++;
19     while(i > 0) { sum += FT[i]; i -= (i & (-i)); }
20     return sum;
21 }

```

```

1 const int n = 10000; // ALL INDICES START AT 1 WITH THIS CODE!!
2
3 // mode 1: update indices, read prefixes
4 void update_idx(int tree[], int i, int val) { // v[i] += val
5     for (; i <= n; i += i & -i) tree[i] += val;
6 }
7 int read_prefix(int tree[], int i) { // get sum v[1..i]
8     int sum = 0;
9     for (; i > 0; i -= i & -i) sum += tree[i];
10    return sum;
11 }
12 int kth(int k) { // find kth element in tree (1-based index)
13     int ans = 0;
14     for (int i = maxl; i >= 0; --i) // maxl = largest i s.t. (1<<i) <= n
15         if (ans + (1<<i) <= n && tree[ans + (1<<i)] < k) {
16             ans += 1<<i;

```

```

17     k -= tree[ans];
18 }
19 return ans+1;
20 }
21
22 // mode 2: update prefixes, read indices
23 void update_prefix(int tree[], int i, int val) { // v[1..i] += val
24     for (; i > 0; i -= i & -i) tree[i] += val;
25 }
26 int read_idx(int tree[], int i) { // get v[i]
27     int sum = 0;
28     for (; i <= n; i += i & -i) sum += tree[i];
29     return sum;
30 }
31
32 // mode 3: range-update range-query
33 const int maxn = 100100;
34 int n;
35 ll mul[maxn], add[maxn];
36
37 void update_idx(ll tree[], int x, ll val) {
38     for (int i = x; i <= n; i += i & -i) tree[i] += val;
39 }
40 void update_prefix(int x, ll val) { // v[x] += val
41     update_idx(mul, 1, val);
42     update_idx(mul, x + 1, -val);
43     update_idx(add, x + 1, x * val);
44 }
45 ll read_prefix(int x) { // get sum v[1..x]
46     ll a = 0, b = 0;
47     for (int i = x; i > 0; i -= i & -i) a += mul[i], b += add[i];
48     return a * x + b;
49 }
50 void update_range(int l, int r, ll val) { // v[l..r] += val
51     update_prefix(l - 1, -val);
52     update_prefix(r, val);
53 }
54 ll read_range(int l, int r) { // get sum v[l..r]
55     return read_prefix(r) - read_prefix(l - 1);
56 }

```

## 1.5 Sparse Table

```

1 struct SparseTable {
2     int st[MAX_N][MAX_LOG + 1], log[MAX_N + 1]; // Achtung: 2^MAX_LOG > MAX_N
3     vector<int> *a;
4
5     // Funktion muss idempotent sein! Hier Minimum.
6     bool better(int lidx, int ridx) { return a->at(lidx) <= a->at(ridx); }
7
8     void init(vector<int> *vec) {
9         a = vec;
10        for (int i = 0; i < (int)a->size(); i++) st[i][0] = i;
11        for (int j = 1; j <= MAX_LOG; j++) {
12            for (int i = 0; i + (1 << j) <= (int)a->size(); i++) {
13                st[i][j] = better(st[i][j - 1], st[i + (1 << (j - 1))][j - 1])
14                    ? st[i][j - 1] : st[i + (1 << (j - 1))][j - 1];

```

```

15     }}
16
17     log[1] = 0;
18     for (int i = 2; i <= MAX_N; i++) log[i] = log[i/2] + 1;
19 }
20
21 // Gibt Index des Ergebnisses in [l,r]. Laufzeit: O(1)
22 int queryIdempotent(int l, int r) {
23     int j = log[r - l + 1];
24     return better(st[l][j], st[r - (1 << j) + 1][j])
25         ? st[l][j] : st[r - (1 << j) + 1][j];
26 }
27 };

```

## 1.6 STL-Tree

```

1 #include <bits/stdc++.h>
2 #include <ext/pb_ds/assoc_container.hpp>
3 #include <ext/pb_ds/tree_policy.hpp>
4 using namespace std; using namespace __gnu_pbds;
5 typedef tree<int, null_type, less<int>, rb_tree_tag,
6     tree_order_statistics_node_update> Tree;
7
8 int main() {
9     Tree X;
10    for (int i = 1; i <= 16; i <= 1) X.insert(i); // {1, 2, 4, 8, 16}
11    cout << *X.find_by_order(3) << endl; // => 8
12    cout << X.order_of_key(10) << endl; // => 4 = min i, mit X[i] >= 10
13    return 0;
14 }

```

## 1.7 STL-Rope (Implicit Cartesian Tree)

```

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

```

## 1.8 Treap (Cartesian Tree)

```

1 struct item {
2     int key, prior;
3     item *l, *r;
4     item() {}
5     item(int key, int prior) : key(key), prior(prior), l(NULL), r(NULL) {}
6 };
7
8 void split(item *t, int key, item *l, item *r) {
9     if (!t) l = r = NULL;
10    else if (key < t->key) split(t->l, key, l, t->l), r = t;
11    else split(t->r, key, t->r, r), l = t;
12 }

```

```

13
14 void insert(item *t, item *it) {
15     if (!t) t = it;
16     else if (it->prior > t->prior) split(t, it->key, it->l, it->r), t = it;
17     else insert(it->key < t->key ? t->l : t->r, it);
18 }
19
20 void merge(item *t, item *l, item *r) {
21     if (!l || !r) t = l ? l : r;
22     else if (l->prior > r->prior) merge(l->r, l->r, r), t = l;
23     else merge(r->l, l, r->l), t = r;
24 }
25
26 void erase(item *t, int key) {
27     if (t->key == key) merge(t, t->l, t->r);
28     else erase(key < t->key ? t->l : t->r, key);
29 }
30
31 item *unite(item *l, item *r) {
32     if (!l || !r) return l ? l : r;
33     if (l->prior < r->prior) swap(l, r);
34     item *lt, rt;
35     split(r, l->key, lt, rt);
36     l->l = unite(l->l, lt);
37     l->r = unite(l->r, rt);
38     return l;
39 }

```

## 1.9 Skew Heap

```

1 // Skew Heap, verschmelzbare Priority Queue.
2 // Laufzeit: Merging, Inserting, DeleteMin: O(log(n)) amortisiert
3 struct node{
4     int key;
5     node *lc, *rc;
6     node(int k) : key(k), lc(0), rc(0) {}
7 } *root = 0;
8 int size = 0;
9
10 node* merge(node *x, node *y) {
11     if (!x) return y;
12     if (!y) return x;
13     if (x->key > y->key) swap(x,y);
14     x->rc = merge(x->rc, y);
15     swap(x->lc, x->rc);
16     return x;
17 }
18
19 void insert(int x) { root = merge(root, new node(x)); size++; }
20
21 int delmin() {
22     if (!root) return -1;
23     int ret = root->key;
24     node *troot = merge(root->lc, root->rc);
25     delete root;
26     root = troot;
27     size--;

```

```

28 return ret;
29 }

```

### 1.10 Lower/Upper Envelop (Convex Hull Optimization)

Um aus einem lower envelope einen upper envelope zu machen (oder umgekehrt), einfach beim Einfügen der Geraden  $m$  und  $b$  negieren.

```

1 // Lower Envelope mit MONOTONEN Inserts und Queries. Jede neue
2 // Gerade hat kleinere Steigung als alle vorherigen.
3 vector<ll> ms, bs; int ptr = 0;
4
5 bool bad(int l1, int l2, int l3) {
6     return (bs[l3]-bs[l1])*(ms[l1]-ms[l2]) < (bs[l2]-bs[l1])*(ms[l1]-ms[l3]);
7 }
8
9 void add(ll m, ll b) { // Laufzeit O(1) amortisiert
10     ms.push_back(m); bs.push_back(b);
11     while (ms.size() >= 3 && bad(ms.size() - 3, ms.size() - 2, ms.size() - 1)) {
12         ms.erase(ms.end() - 2); bs.erase(bs.end() - 2);
13     }
14
15 ll get(int idx, ll x) { return ms[idx] * x + bs[idx]; }
16
17 ll query(ll x) { // Laufzeit: O(1) amortisiert
18     if (ptr >= (int)ms.size()) ptr = ms.size() - 1;
19     while (ptr < (int)ms.size() - 1 && get(ptr + 1, x) < get(ptr, x)) ptr++;
20     return ms[ptr] * x + bs[ptr];
21 }

```

```

1 // Upper Envelope, dynamisch.
2 struct Line {
3     ll m, b;
4     mutable function<const Line*> succ;
5     bool operator<(const Line& rhs) const {
6         if (rhs.b != is_query) return m < rhs.m;
7         const Line* s = succ();
8         if (!s) return 0;
9         ll x = rhs.m;
10        return b - s->b < (s->m - m) * x;
11    }
12 };
13 struct HullDynamic : public multiset<Line> {
14     bool bad(iterator y) {
15         auto z = next(y);
16         if (y == begin()) {
17             if (z == end()) return 0;
18             return y->m == z->m && y->b <= z->b;
19         }
20         auto x = prev(y);
21         if (z == end()) return y->m == x->m && y->b <= x->b;
22         return (x->b - y->b)*(z->m - y->m) >= (y->b - z->b)*(y->m - x->m);
23     }
24     void insert_line(ll m, ll b) { // Laufzeit: O(log(n))
25         auto y = insert({ m, b });
26         y->succ = [=] { return next(y) == end() ? 0 : *next(y); };

```

```

27         if (bad(y)) { erase(y); return; }
28         while (next(y) != end() && bad(next(y))) erase(next(y));
29         while (y != begin() && bad(prev(y))) erase(prev(y));
30     }
31     ll query(ll x) { // Laufzeit: O(log(n))
32         auto l = *lower_bound((Line) { x, -(1LL<<62) });
33         return l.m * x + l.b;
34     }
35 };

```

## 2 Graphen

### 2.1 Kürzeste Wege

```

1 // Laufzeit: O((|E|+|V|)*log |V|)
2 void dijkstra(int start) {
3     priority_queue<ii, vector<ii>, greater<ii> > pq;
4     vector<int> dist(NUM_VERTICES, INF), parent(NUM_VERTICES, -1);
5     dist[start] = 0; pq.push(ii(0, start));
6
7     while (!pq.empty()) {
8         ii front = pq.top(); pq.pop();
9         int curNode = front.second, curDist = front.first;
10        if (curDist > dist[curNode]) continue; // WICHTIG!
11
12        for (auto n : adjlist[curNode]) {
13            int nextNode = n.first, nextDist = curDist + n.second;
14            if (nextDist < dist[nextNode]) {
15                dist[nextNode] = nextDist; parent[nextNode] = curNode;
16                pq.push(ii(nextDist, nextNode));
17            }

```

```

1 // Laufzeit: O(|V|*|E|)
2 vector<edge> edges; // Kanten einfügen!
3 vector<int> dist, parent;
4
5 void bellmannFord() {
6     dist.assign(NUM_VERTICES, INF); dist[0] = 0;
7     parent.assign(NUM_VERTICES, -1);
8     for (int i = 0; i < NUM_VERTICES - 1; i++) {
9         for (auto &e : edges) {
10            if (dist[e.from] + e.cost < dist[e.to]) {
11                dist[e.to] = dist[e.from] + e.cost;
12                parent[e.to] = e.from;
13            }
14        }
15
16        // "dist" und "parent" sind korrekte kürzeste Pfade.
17        // Folgende Zeilen prüfen nur negative Kreise.
18        for (auto &e : edges) {
19            if (dist[e.from] + e.cost < dist[e.to]) {
20                // Negativer Kreis gefunden.

```

Floyd Warshall:

- Nur negative Werte sollten die Nullen bei Schlingen überschreiben.
- Von parallelen Kanten sollte nur die günstigste gespeichert werden.

- $i$  liegt genau dann auf einem negativen Kreis, wenn  $\text{dist}[i][i] < 0$  ist.
- Wenn für  $c$  gilt, dass  $\text{dist}[u][c] \neq \text{INF} \ \&\& \ \text{dist}[c][v] \neq \text{INF} \ \&\& \ \text{dist}[c][c] < 0$ , wird der  $u$ - $v$ -Pfad beliebig kurz.

## 2.2 Strongly Connected Components (TARJANS-Algorithmus)

```

1 // Laufzeit:  $O(|V|+|E|)$ 
2 int counter, sccCounter;
3 vector<bool> visited, inStack;
4 vector<vector<int>> adjlist;
5 vector<int> d, low, sccs; // sccs enthält den Index der SCC pro Knoten.
6 stack<int> s;
7
8 void visit(int v) {
9     visited[v] = true;
10    d[v] = low[v] = counter++;
11    s.push(v); inStack[v] = true;
12
13    for (auto u : adjlist[v]) {
14        if (!visited[u]) {
15            visit(u);
16            low[v] = min(low[v], low[u]);
17        } else if (inStack[u]) {
18            low[v] = min(low[v], low[u]);
19        }
20    }
21
22    if (d[v] == low[v]) {
23        int u;
24        do {
25            u = s.top(); s.pop(); inStack[u] = false;
26            sccs[u] = sccCounter;
27        } while (u != v);
28        sccCounter++;
29    }
30
31    void scc() {
32        visited.assign(adjlist.size(), false);
33        d.assign(adjlist.size(), -1);
34        low.assign(adjlist.size(), -1);
35        inStack.assign(adjlist.size(), false);
36        sccs.resize(adjlist.size(), -1);
37
38        counter = sccCounter = 0;
39        for (int i = 0; i < (int)adjlist.size(); i++) {
40            if (!visited[i]) {
41                visit(i);
42            }
43        }
44    }
45
46    void dfs(int v, int parent = -1) {
47        d[v] = low[v] = ++counter;
48        if (parent == root) ++rootCount;
49
50        for (auto w : adjlist[v]) {
51            if (!d[w]) {
52                dfs(w, v);
53                if (low[w] >= d[v] && v != root) isArt[v] = true;
54                if (low[w] > d[v]) bridges.push_back(ii(v, w));
55                low[v] = min(low[v], low[w]);
56            } else if (w != parent) {
57                low[v] = min(low[v], d[w]);
58            }
59        }
60    }
61
62    void findArticulationPoints() {
63        counter = 0;
64        low.resize(adjlist.size());
65        d.assign(adjlist.size(), 0);
66        isArt.assign(adjlist.size(), false);
67        bridges.clear(); // nur fuer Bruecken
68        for (int v = 0; v < (int)adjlist.size(); v++) {
69            if (!d[v]) {
70                root = v; rootCount = 0;
71                dfs(v);
72                if (rootCount > 1) isArt[v] = true;
73            }
74        }
75    }
76
77    void scc() {
78        visited.assign(adjlist.size(), false);
79        d.assign(adjlist.size(), -1);
80        low.assign(adjlist.size(), -1);
81        inStack.assign(adjlist.size(), false);
82        sccs.resize(adjlist.size(), -1);
83
84        counter = sccCounter = 0;
85        for (int i = 0; i < (int)adjlist.size(); i++) {
86            if (!visited[i]) {
87                visit(i);
88            }
89        }
90    }
91
92    void swapEdges(int n, int a, int b) {
93        int neighA = adjlist[n][a], neighB = adjlist[n][b];
94        adjlist[n][a] = b;
95        adjlist[n][b] = a;
96    }
97
98    void dfs(int v, int parent = -1) {
99        d[v] = low[v] = ++counter;
100       if (parent == root) ++rootCount;
101
102       for (auto w : adjlist[v]) {
103           if (!d[w]) {
104               dfs(w, v);
105               if (low[w] >= d[v] && v != root) isArt[v] = true;
106               if (low[w] > d[v]) bridges.push_back(ii(v, w));
107               low[v] = min(low[v], low[w]);
108           } else if (w != parent) {
109               low[v] = min(low[v], d[w]);
110           }
111       }
112   }
113
114   void findArticulationPoints() {
115       counter = 0;
116       low.resize(adjlist.size());
117       d.assign(adjlist.size(), 0);
118       isArt.assign(adjlist.size(), false);
119       bridges.clear(); // nur fuer Bruecken
120       for (int v = 0; v < (int)adjlist.size(); v++) {
121           if (!d[v]) {
122               root = v; rootCount = 0;
123               dfs(v);
124               if (rootCount > 1) isArt[v] = true;
125           }
126       }
127   }
128
129   void scc() {
130       visited.assign(adjlist.size(), false);
131       d.assign(adjlist.size(), -1);
132       low.assign(adjlist.size(), -1);
133       inStack.assign(adjlist.size(), false);
134       sccs.resize(adjlist.size(), -1);
135
136       counter = sccCounter = 0;
137       for (int i = 0; i < (int)adjlist.size(); i++) {
138           if (!visited[i]) {
139               visit(i);
140           }
141       }
142   }
143
144   void swapEdges(int n, int a, int b) {
145       int neighA = adjlist[n][a], neighB = adjlist[n][b];
146       adjlist[n][a] = b;
147       adjlist[n][b] = a;
148   }
149
150   void dfs(int v, int parent = -1) {
151       d[v] = low[v] = ++counter;
152       if (parent == root) ++rootCount;
153
154       for (auto w : adjlist[v]) {
155           if (!d[w]) {
156               dfs(w, v);
157               if (low[w] >= d[v] && v != root) isArt[v] = true;
158               if (low[w] > d[v]) bridges.push_back(ii(v, w));
159               low[v] = min(low[v], low[w]);
160           } else if (w != parent) {
161               low[v] = min(low[v], d[w]);
162           }
163       }
164   }
165
166   void findArticulationPoints() {
167       counter = 0;
168       low.resize(adjlist.size());
169       d.assign(adjlist.size(), 0);
170       isArt.assign(adjlist.size(), false);
171       bridges.clear(); // nur fuer Bruecken
172       for (int v = 0; v < (int)adjlist.size(); v++) {
173           if (!d[v]) {
174               root = v; rootCount = 0;
175               dfs(v);
176               if (rootCount > 1) isArt[v] = true;
177           }
178       }
179   }
180
181   void scc() {
182       visited.assign(adjlist.size(), false);
183       d.assign(adjlist.size(), -1);
184       low.assign(adjlist.size(), -1);
185       inStack.assign(adjlist.size(), false);
186       sccs.resize(adjlist.size(), -1);
187
188       counter = sccCounter = 0;
189       for (int i = 0; i < (int)adjlist.size(); i++) {
190           if (!visited[i]) {
191               visit(i);
192           }
193       }
194   }
195
196   void swapEdges(int n, int a, int b) {
197       int neighA = adjlist[n][a], neighB = adjlist[n][b];
198       adjlist[n][a] = b;
199       adjlist[n][b] = a;
200   }
201
202   void dfs(int v, int parent = -1) {
203       d[v] = low[v] = ++counter;
204       if (parent == root) ++rootCount;
205
206       for (auto w : adjlist[v]) {
207           if (!d[w]) {
208               dfs(w, v);
209               if (low[w] >= d[v] && v != root) isArt[v] = true;
210               if (low[w] > d[v]) bridges.push_back(ii(v, w));
211               low[v] = min(low[v], low[w]);
212           } else if (w != parent) {
213               low[v] = min(low[v], d[w]);
214           }
215       }
216   }
217
218   void findArticulationPoints() {
219       counter = 0;
220       low.resize(adjlist.size());
221       d.assign(adjlist.size(), 0);
222       isArt.assign(adjlist.size(), false);
223       bridges.clear(); // nur fuer Bruecken
224       for (int v = 0; v < (int)adjlist.size(); v++) {
225           if (!d[v]) {
226               root = v; rootCount = 0;
227               dfs(v);
228               if (rootCount > 1) isArt[v] = true;
229           }
230       }
231   }
232
233   void scc() {
234       visited.assign(adjlist.size(), false);
235       d.assign(adjlist.size(), -1);
236       low.assign(adjlist.size(), -1);
237       inStack.assign(adjlist.size(), false);
238       sccs.resize(adjlist.size(), -1);
239
240       counter = sccCounter = 0;
241       for (int i = 0; i < (int)adjlist.size(); i++) {
242           if (!visited[i]) {
243               visit(i);
244           }
245       }
246   }
247
248   void swapEdges(int n, int a, int b) {
249       int neighA = adjlist[n][a], neighB = adjlist[n][b];
250       adjlist[n][a] = b;
251       adjlist[n][b] = a;
252   }
253
254   void dfs(int v, int parent = -1) {
255       d[v] = low[v] = ++counter;
256       if (parent == root) ++rootCount;
257
258       for (auto w : adjlist[v]) {
259           if (!d[w]) {
260               dfs(w, v);
261               if (low[w] >= d[v] && v != root) isArt[v] = true;
262               if (low[w] > d[v]) bridges.push_back(ii(v, w));
263               low[v] = min(low[v], low[w]);
264           } else if (w != parent) {
265               low[v] = min(low[v], d[w]);
266           }
267       }
268   }
269
270   void findArticulationPoints() {
271       counter = 0;
272       low.resize(adjlist.size());
273       d.assign(adjlist.size(), 0);
274       isArt.assign(adjlist.size(), false);
275       bridges.clear(); // nur fuer Bruecken
276       for (int v = 0; v < (int)adjlist.size(); v++) {
277           if (!d[v]) {
278               root = v; rootCount = 0;
279               dfs(v);
280               if (rootCount > 1) isArt[v] = true;
281           }
282       }
283   }
284
285   void scc() {
286       visited.assign(adjlist.size(), false);
287       d.assign(adjlist.size(), -1);
288       low.assign(adjlist.size(), -1);
289       inStack.assign(adjlist.size(), false);
290       sccs.resize(adjlist.size(), -1);
291
292       counter = sccCounter = 0;
293       for (int i = 0; i < (int)adjlist.size(); i++) {
294           if (!visited[i]) {
295               visit(i);
296           }
297       }
298   }
299
300   void swapEdges(int n, int a, int b) {
301       int neighA = adjlist[n][a], neighB = adjlist[n][b];
302       adjlist[n][a] = b;
303       adjlist[n][b] = a;
304   }
305
306   void dfs(int v, int parent = -1) {
307       d[v] = low[v] = ++counter;
308       if (parent == root) ++rootCount;
309
310       for (auto w : adjlist[v]) {
311           if (!d[w]) {
312               dfs(w, v);
313               if (low[w] >= d[v] && v != root) isArt[v] = true;
314               if (low[w] > d[v]) bridges.push_back(ii(v, w));
315               low[v] = min(low[v], low[w]);
316           } else if (w != parent) {
317               low[v] = min(low[v], d[w]);
318           }
319       }
320   }
321
322   void findArticulationPoints() {
323       counter = 0;
324       low.resize(adjlist.size());
325       d.assign(adjlist.size(), 0);
326       isArt.assign(adjlist.size(), false);
327       bridges.clear(); // nur fuer Bruecken
328       for (int v = 0; v < (int)adjlist.size(); v++) {
329           if (!d[v]) {
330               root = v; rootCount = 0;
331               dfs(v);
332               if (rootCount > 1) isArt[v] = true;
333           }
334       }
335   }
336
337   void scc() {
338       visited.assign(adjlist.size(), false);
339       d.assign(adjlist.size(), -1);
340       low.assign(adjlist.size(), -1);
341       inStack.assign(adjlist.size(), false);
342       sccs.resize(adjlist.size(), -1);
343
344       counter = sccCounter = 0;
345       for (int i = 0; i < (int)adjlist.size(); i++) {
346           if (!visited[i]) {
347               visit(i);
348           }
349       }
350   }
351
352   void swapEdges(int n, int a, int b) {
353       int neighA = adjlist[n][a], neighB = adjlist[n][b];
354       adjlist[n][a] = b;
355       adjlist[n][b] = a;
356   }
357
358   void dfs(int v, int parent = -1) {
359       d[v] = low[v] = ++counter;
360       if (parent == root) ++rootCount;
361
362       for (auto w : adjlist[v]) {
363           if (!d[w]) {
364               dfs(w, v);
365               if (low[w] >= d[v] && v != root) isArt[v] = true;
366               if (low[w] > d[v]) bridges.push_back(ii(v, w));
367               low[v] = min(low[v], low[w]);
368           } else if (w != parent) {
369               low[v] = min(low[v], d[w]);
370           }
371       }
372   }
373
374   void findArticulationPoints() {
375       counter = 0;
376       low.resize(adjlist.size());
377       d.assign(adjlist.size(), 0);
378       isArt.assign(adjlist.size(), false);
379       bridges.clear(); // nur fuer Bruecken
380       for (int v = 0; v < (int)adjlist.size(); v++) {
381           if (!d[v]) {
382               root = v; rootCount = 0;
383               dfs(v);
384               if (rootCount > 1) isArt[v] = true;
385           }
386       }
387   }
388
389   void scc() {
390       visited.assign(adjlist.size(), false);
391       d.assign(adjlist.size(), -1);
392       low.assign(adjlist.size(), -1);
393       inStack.assign(adjlist.size(), false);
394       sccs.resize(adjlist.size(), -1);
395
396       counter = sccCounter = 0;
397       for (int i = 0; i < (int)adjlist.size(); i++) {
398           if (!visited[i]) {
399               visit(i);
400           }
401       }
402   }
403
404   void swapEdges(int n, int a, int b) {
405       int neighA = adjlist[n][a], neighB = adjlist[n][b];
406       adjlist[n][a] = b;
407       adjlist[n][b] = a;
408   }
409
410   void dfs(int v, int parent = -1) {
411       d[v] = low[v] = ++counter;
412       if (parent == root) ++rootCount;
413
414       for (auto w : adjlist[v]) {
415           if (!d[w]) {
416               dfs(w, v);
417               if (low[w] >= d[v] && v != root) isArt[v] = true;
418               if (low[w] > d[v]) bridges.push_back(ii(v, w));
419               low[v] = min(low[v], low[w]);
420           } else if (w != parent) {
421               low[v] = min(low[v], d[w]);
422           }
423       }
424   }
425
426   void findArticulationPoints() {
427       counter = 0;
428       low.resize(adjlist.size());
429       d.assign(adjlist.size(), 0);
430       isArt.assign(adjlist.size(), false);
431       bridges.clear(); // nur fuer Bruecken
432       for (int v = 0; v < (int)adjlist.size(); v++) {
433           if (!d[v]) {
434               root = v; rootCount = 0;
435               dfs(v);
436               if (rootCount > 1) isArt[v] = true;
437           }
438       }
439   }
440
441   void scc() {
442       visited.assign(adjlist.size(), false);
443       d.assign(adjlist.size(), -1);
444       low.assign(adjlist.size(), -1);
445       inStack.assign(adjlist.size(), false);
446       sccs.resize(adjlist.size(), -1);
447
448       counter = sccCounter = 0;
449       for (int i = 0; i < (int)adjlist.size(); i++) {
450           if (!visited[i]) {
451               visit(i);
452           }
453       }
454   }
455
456   void swapEdges(int n, int a, int b) {
457       int neighA = adjlist[n][a], neighB = adjlist[n][b];
458       adjlist[n][a] = b;
459       adjlist[n][b] = a;
460   }
461
462   void dfs(int v, int parent = -1) {
463       d[v] = low[v] = ++counter;
464       if (parent == root) ++rootCount;
465
466       for (auto w : adjlist[v]) {
467           if (!d[w]) {
468               dfs(w, v);
469               if (low[w] >= d[v] && v != root) isArt[v] = true;
470               if (low[w] > d[v]) bridges.push_back(ii(v, w));
471               low[v] = min(low[v], low[w]);
472           } else if (w != parent) {
473               low[v] = min(low[v], d[w]);
474           }
475       }
476   }
477
478   void findArticulationPoints() {
479       counter = 0;
480       low.resize(adjlist.size());
481       d.assign(adjlist.size(), 0);
482       isArt.assign(adjlist.size(), false);
483       bridges.clear(); // nur fuer Bruecken
484       for (int v = 0; v < (int)adjlist.size(); v++) {
485           if (!d[v]) {
486               root = v; rootCount = 0;
487               dfs(v);
488               if (rootCount > 1) isArt[v] = true;
489           }
490       }
491   }
492
493   void scc() {
494       visited.assign(adjlist.size(), false);
495       d.assign(adjlist.size(), -1);
496       low.assign(adjlist.size(), -1);
497       inStack.assign(adjlist.size(), false);
498       sccs.resize(adjlist.size(), -1);
499
500       counter = sccCounter = 0;
501       for (int i = 0; i < (int)adjlist.size(); i++) {
502           if (!visited[i]) {
503               visit(i);
504           }
505       }
506   }
507
508   void swapEdges(int n, int a, int b) {
509       int neighA = adjlist[n][a], neighB = adjlist[n][b];
510       adjlist[n][a] = b;
511       adjlist[n][b] = a;
512   }
513
514   void dfs(int v, int parent = -1) {
515       d[v] = low[v] = ++counter;
516       if (parent == root) ++rootCount;
517
518       for (auto w : adjlist[v]) {
519           if (!d[w]) {
520               dfs(w, v);
521               if (low[w] >= d[v] && v != root) isArt[v] = true;
522               if (low[w] > d[v]) bridges.push_back(ii(v, w));
523               low[v] = min(low[v], low[w]);
524           } else if (w != parent) {
525               low[v] = min(low[v], d[w]);
526           }
527       }
528   }
529
530   void findArticulationPoints() {
531       counter = 0;
532       low.resize(adjlist.size());
533       d.assign(adjlist.size(), 0);
534       isArt.assign(adjlist.size(), false);
535       bridges.clear(); // nur fuer Bruecken
536       for (int v = 0; v < (int)adjlist.size(); v++) {
537           if (!d[v]) {
538               root = v; rootCount = 0;
539               dfs(v);
540               if (rootCount > 1) isArt[v] = true;
541           }
542       }
543   }
544
545   void scc() {
546       visited.assign(adjlist.size(), false);
547       d.assign(adjlist.size(), -1);
548       low.assign(adjlist.size(), -1);
549       inStack.assign(adjlist.size(), false);
550       sccs.resize(adjlist.size(), -1);
551
552       counter = sccCounter = 0;
553       for (int i = 0; i < (int)adjlist.size(); i++) {
554           if (!visited[i]) {
555               visit(i);
556           }
557       }
558   }
559
560   void swapEdges(int n, int a, int b) {
561       int neighA = adjlist[n][a], neighB = adjlist[n][b];
562       adjlist[n][a] = b;
563       adjlist[n][b] = a;
564   }
565
566   void dfs(int v, int parent = -1) {
567       d[v] = low[v] = ++counter;
568       if (parent == root) ++rootCount;
569
570       for (auto w : adjlist[v]) {
571           if (!d[w]) {
572               dfs(w, v);
573               if (low[w] >= d[v] && v != root) isArt[v] = true;
574               if (low[w] > d[v]) bridges.push_back(ii(v, w));
575               low[v] = min(low[v], low[w]);
576           } else if (w != parent) {
577               low[v] = min(low[v], d[w]);
578           }
579       }
580   }
581
582   void findArticulationPoints() {
583       counter = 0;
584       low.resize(adjlist.size());
585       d.assign(adjlist.size(), 0);
586       isArt.assign(adjlist.size(), false);
587       bridges.clear(); // nur fuer Bruecken
588       for (int v = 0; v < (int)adjlist.size(); v++) {
589           if (!d[v]) {
590               root = v; rootCount = 0;
591               dfs(v);
592               if (rootCount > 1) isArt[v] = true;
593           }
594       }
595   }
596
597   void scc() {
598       visited.assign(adjlist.size(), false);
599       d.assign(adjlist.size(), -1);
600       low.assign(adjlist.size(), -1);
601       inStack.assign(adjlist.size(), false);
602       sccs.resize(adjlist.size(), -1);
603
604       counter = sccCounter = 0;
605       for (int i = 0; i < (int)adjlist.size(); i++) {
606           if (!visited[i]) {
607               visit(i);
608           }
609       }
610   }
611
612   void swapEdges(int n, int a, int b) {
613       int neighA = adjlist[n][a], neighB = adjlist[n][b];
614       adjlist[n][a] = b;
615       adjlist[n][b] = a;
616   }
617
618   void dfs(int v, int parent = -1) {
619       d[v] = low[v] = ++counter;
620       if (parent == root) ++rootCount;
621
622       for (auto w : adjlist[v]) {
623           if (!d[w]) {
624               dfs(w, v);
625               if (low[w] >= d[v] && v != root) isArt[v] = true;
626               if (low[w] > d[v]) bridges.push_back(ii(v, w));
627               low[v] = min(low[v], low[w]);
628           } else if (w != parent) {
629               low[v] = min(low[v], d[w]);
630           }
631       }
632   }
633
634   void findArticulationPoints() {
635       counter = 0;
636       low.resize(adjlist.size());
637       d.assign(adjlist.size(), 0);
638       isArt.assign(adjlist.size(), false);
639       bridges.clear(); // nur fuer Bruecken
640       for (int v = 0; v < (int)adjlist.size(); v++) {
641           if (!d[v]) {
642               root = v; rootCount = 0;
643               dfs(v);
644               if (rootCount > 1) isArt[v] = true;
645           }
646       }
647   }
648
649   void scc() {
650       visited.assign(adjlist.size(), false);
651       d.assign(adjlist.size(), -1);
652       low.assign(adjlist.size(), -1);
653       inStack.assign(adjlist.size(), false);
654       sccs.resize(adjlist.size(), -1);
655
656       counter = sccCounter = 0;
657       for (int i = 0; i < (int)adjlist.size(); i++) {
658           if (!visited[i]) {
659               visit(i);
660           }
661       }
662   }
663
664   void swapEdges(int n, int a, int b) {
665       int neighA = adjlist[n][a], neighB = adjlist[n][b];
666       adjlist[n][a] = b;
667       adjlist[n][b] = a;
668   }
669
670   void dfs(int v, int parent = -1) {
671       d[v] = low[v] = ++counter;
672       if (parent == root) ++rootCount;
673
674       for (auto w : adjlist[v]) {
675           if (!d[w]) {
676               dfs(w, v);
677               if (low[w] >= d[v] && v != root) isArt[v] = true;
678               if (low[w] > d[v]) bridges.push_back(ii(v, w));
679               low[v] = min(low[v], low[w]);
680           } else if (w != parent) {
681               low[v] = min(low[v], d[w]);
682           }
683       }
684   }
685
686   void findArticulationPoints() {
687       counter = 0;
688       low.resize(adjlist.size());
689       d.assign(adjlist.size(), 0);
690       isArt.assign(adjlist.size(), false);
691       bridges.clear(); // nur fuer Bruecken
692       for (int v = 0; v < (int)adjlist.size(); v++) {
693           if (!d[v]) {
694               root = v; rootCount = 0;
695               dfs(v);
696               if (rootCount > 1) isArt[v] = true;
697           }
698       }
699   }
700
701   void scc() {
702       visited.assign(adjlist.size(), false);
703       d.assign(adjlist.size(), -1);
704       low.assign(adjlist.size(), -1);
705       inStack.assign(adjlist.size(), false);
706       sccs.resize(adjlist.size(), -1);
707
708       counter = sccCounter = 0;
709       for (int i = 0; i < (int)adjlist.size(); i++) {
710           if (!visited[i]) {
711               visit(i);
712           }
713       }
714   }
715
716   void swapEdges(int n, int a, int b) {
717       int neighA = adjlist[n][a], neighB = adjlist[n][b];
718       adjlist[n][a] = b;
719       adjlist[n][b] = a;
720   }
721
722   void dfs(int v, int parent = -1) {
723       d[v] = low[v] = ++counter;
724       if (parent == root) ++rootCount;
725
726       for (auto w : adjlist[v]) {
727           if (!d[w]) {
728               dfs(w, v);
729               if (low[w] >= d[v] && v != root) isArt[v] = true;
730               if (low[w] > d[v]) bridges.push_back(ii(v, w));
731               low[v] = min(low[v], low[w]);
732           } else if (w != parent) {
733               low[v] = min(low[v], d[w]);
734           }
735       }
736   }
737
738   void findArticulationPoints() {
739       counter = 0;
740       low.resize(adjlist.size());
741       d.assign(adjlist.size(), 0);
742       isArt.assign(adjlist.size(), false);
743       bridges.clear(); // nur fuer Bruecken
744       for (int v = 0; v < (int)adjlist.size(); v++) {
745           if (!d[v]) {
746               root = v; rootCount = 0;
747               dfs(v);
748               if (rootCount > 1) isArt[v] = true;
749           }
750       }
751   }
752
753   void scc() {
754       visited.assign(adjlist.size(), false);
755       d.assign(adjlist.size(), -1);
756       low.assign(adjlist.size(), -1);
757       inStack.assign(adjlist.size(), false);
758       sccs.resize(adjlist.size(), -1);
759
760       counter = sccCounter = 0;
761       for (int i = 0; i < (int)adjlist.size(); i++) {
762           if (!visited[i]) {
763               visit(i);
764           }
765       }
766   }
767
768   void swapEdges(int n, int a, int b) {
769       int neighA = adjlist[n][a], neighB = adjlist[n][b];
770       adjlist[n][a] = b;
771       adjlist[n][b] = a;
772   }
773
774   void dfs(int v, int parent = -1) {
775       d[v] = low[v] = ++counter;
776       if (parent == root) ++rootCount;
777
778       for (auto w : adjlist[v]) {
779           if (!d[w]) {
780               dfs(w, v);
781               if (low[w] >= d[v] && v != root) isArt[v] = true;
782               if (low[w] > d[v]) bridges.push_back(ii(v, w));
783               low[v] = min(low[v], low[w]);
784           } else if (w != parent) {
785               low[v] = min(low[v], d[w]);
786           }
787       }
788   }
789
790   void findArticulationPoints() {
791       counter = 0;
792       low.resize(adjlist.size());
793       d.assign(adjlist.size(), 0);
794       isArt.assign(adjlist.size(), false);
795       bridges.clear(); // nur fuer Bruecken
796       for (int v = 0; v < (int)adjlist.size(); v++) {
797           if (!d[v]) {
798               root = v; rootCount = 0;
799               dfs(v);
800               if (rootCount > 1) isArt[v] = true;
801           }
802       }
803   }
804
805   void scc() {
806       visited.assign(adjlist.size(), false);
807       d.assign(adjlist.size(), -1);
808       low.assign(adjlist.size(), -1);
809       inStack.assign(adjlist.size(), false);
810       sccs.resize(adjlist.size(), -1);
811
812       counter = sccCounter = 0;
813       for (int i = 0; i < (int)adjlist.size(); i++) {
814           if (!visited[i]) {
815               visit(i);
816           }
817       }
818   }
819
820   void swapEdges(int n, int a, int b) {
821       int neighA = adjlist[n][a], neighB = adjlist[n][b];
822       adjlist[n][a] = b;
823       adjlist[n][b] = a;
824   }
825
826   void dfs(int v, int parent = -1) {
827       d[v] = low[v] = ++counter;
828       if (parent == root) ++rootCount;
829
830       for (auto w : adjlist[v]) {
831           if (!d[w]) {
832               dfs(w, v);
833               if (low[w] >= d[v] && v != root) isArt[v] = true;
834               if (low[w] > d[v]) bridges.push_back(ii(v, w));
835               low[v] = min(low[v], low[w]);
836           } else if (w != parent) {
837               low[v] = min(low[v], d[w]);
838           }
839       }
840   }
841
842   void findArticulationPoints() {
843       counter = 0;
844       low.resize(adjlist.size());
845       d.assign(adjlist.size(), 0);
846       isArt.assign(adjlist.size(), false);
847       bridges.clear(); // nur fuer Bruecken
848       for (int v = 0; v < (int)adjlist.size(); v++) {
849           if (!d[v]) {
850               root = v; rootCount = 0;
851               dfs(v);
852               if (rootCount > 1) isArt[v] = true;
853           }
854       }
855   }
856
857   void scc() {
858       visited.assign(adjlist.size(), false);
859       d.assign(adjlist.size(), -1);
860       low.assign(adjlist.size(), -1);
861       inStack.assign(adjlist.size(), false);
862       sccs.resize(adjlist.size(), -1);
863
864       counter = sccCounter = 0;
865       for (int i = 0; i < (int)adjlist.size(); i++) {
866           if (!visited[i]) {
867               visit(i);
868           }
869       }
870   }
871
872   void swapEdges(int n, int a, int b) {
873       int neighA = adjlist[n][a], neighB = adjlist[n][b];
874       adjlist[n][a] = b;
875       adjlist[n][b] = a;
876   }
877
878   void dfs(int v, int parent = -1) {
879       d[v] = low[v] = ++counter;
880       if (parent == root) ++rootCount;
881
882       for (auto w : adjlist[v]) {
883           if (!d[w]) {
884               dfs(w, v);
885               if (low[w] >= d[v] && v != root) isArt[v] = true;
886               if (low[w] > d[v]) bridges.push_back(ii(v, w));
887               low[v] = min(low[v], low[w]);
888           } else if (w != parent) {
889               low[v] = min(low[v], d[w]);
890           }
891       }
892   }
893
894   void findArticulationPoints() {
895       counter = 0;
896       low.resize(adjlist.size());
897       d.assign(adjlist.size(), 0);
898       isArt.assign(adjlist.size(), false);
899       bridges.clear(); // nur fuer Bruecken
900       for (int v = 0; v < (int)adjlist.size(); v++) {
901           if (!d[v]) {
902               root = v; rootCount = 0;
903               dfs(v);
904               if (rootCount > 1) isArt[v] = true;
905           }
906       }
907   }
908
909   void scc() {
910       visited.assign(adjlist.size(), false);
911       d.assign(adjlist.size(), -1);
912       low.assign(adjlist.size(), -1);
913       inStack.assign(adjlist.size(), false);
914       sccs.resize(adjlist.size(), -1);
915
916       counter = sccCounter = 0;
917       for (int i = 0; i < (int)adjlist.size(); i++) {
918           if (!visited[i]) {
919               visit(i);
920           }
921       }
922   }
923
924   void swapEdges(int n, int a, int b) {
925       int neighA = adjlist[n][a], neighB = adjlist[n][b];
926       adjlist[n][a] = b;
927       adjlist[n][b] = a;
928   }
929
930   void dfs(int v, int parent = -1) {
931       d[v] = low[v] = ++counter;
932       if (parent == root) ++rootCount;
933
934       for (auto w : adjlist[v]) {
935           if (!d[w]) {
936               dfs(w, v);
937               if (low[w] >= d[v] && v != root) isArt[v] = true;
938               if (low[w] > d[v]) bridges.push_back(ii(v, w));
939               low[v] = min(low[v], low[w]);
940           } else if (w != parent) {
941               low[v] = min(low[v], d[w]);
942           }
943       }
944   }
945
946   void findArticulationPoints() {
947       counter = 0;
948       low.resize(adjlist.size());
949       d.assign(adjlist.size(), 0);
950       isArt.assign(adjlist.size(), false);
951       bridges.clear(); // nur fuer Bruecken
952       for (int v = 0; v < (int)adjlist.size(); v++) {
953           if (!d[v]) {
954               root = v; rootCount = 0;
955               dfs(v);
956               if (rootCount > 1) isArt[v] = true;
957           }
958       }
959   }
960
961   void scc() {
962       visited.assign(adjlist.size(), false);
963       d.assign(adjlist.size(), -1);
964       low.assign(adjlist.size(), -1);
965       inStack.assign(adjlist.size(), false);
966       sccs.resize(adjlist.size(), -1);
967
968       counter = sccCounter = 0;
969       for (int i = 0; i < (int)adjlist.size(); i++) {
970           if (!visited[i]) {
971               visit(i);
972           }
973       }
974   }
975
976   void swapEdges(int n, int a, int b) {
977       int neighA = adjlist[n][a], neighB = adjlist[n][b];
978       adjlist[n][a] = b;
979       adjlist[n][b] = a;
980   }
981
982   void dfs(int v, int parent = -1) {
983       d[v] = low[v] = ++counter;
984       if (parent == root) ++rootCount;
985
986
```

```

8   int idxNeighA = otherIdx[n][a], idxNeighB = otherIdx[n][b];
9   swap(adjlist[n][a], adjlist[n][b]);
10  swap(otherIdx[n][a], otherIdx[n][b]);
11  otherIdx[neighA][idxNeighA] = b;
12  otherIdx[neighB][idxNeighB] = a;
13 }
14
15 // Entfernt Kante i von Knoten n (und die zugehörige Rückwärtskante).
16 void removeEdge(int n, int i) {
17     int other = adjlist[n][i];
18     if (other == n) { //Schlingen.
19         validIdx[n]++;
20         return;
21     }
22     int otherIndex = otherIdx[n][i];
23     validIdx[n]++;
24     if (otherIndex != validIdx[other]) {
25         swapEdges(other, otherIndex, validIdx[other]);
26     }
27     validIdx[other]++;
28 }
29
30 // Findet Eulerzyklus an Knoten n startend.
31 // Teste vorher, dass Graph zusammenhängend ist! Isolierten Knoten?
32 // Teste vorher, ob Eulerzyklus überhaupt existiert!
33 void euler(int n) {
34     while (validIdx[n] < (int)adjlist[n].size()) {
35         int nn = adjlist[n][validIdx[n]];
36         removeEdge(n, validIdx[n]);
37         euler(nn);
38     }
39     cycle.push_back(n); // Zyklus in cycle in umgekehrter Reihenfolge.
40 }

```

## 2.5 Lowest Common Ancestor

```

1 struct LCA {
2     vector<int> depth, visited, first;
3     int idx;
4     SparseTable st;
5
6     void init(vector<vector<int>> &g, int root) { // Laufzeit: O(|V|)
7         depth.assign(2 * g.size(), 0);
8         visited.assign(2 * g.size(), -1);
9         first.assign(g.size(), 2 * g.size());
10        idx = 0;
11        visit(g, root, 0);
12        st.init(&depth);
13    }
14
15    void visit(vector<vector<int>> &g, int v, int d) {
16        visited[idx] = v, depth[idx] = d, first[v] = min(idx, first[v]), idx++;
17
18        for (int w : g[v]) {
19            if (first[w] == 2 * (int)g.size()) {
20                visit(g, w, d + 1);
21                visited[idx] = v, depth[idx] = d, idx++;

```

```

22    }
23
24    int getLCA(int a, int b) { // Laufzeit: O(1)
25        if (first[a] > first[b]) swap(a, b);
26        return visited[st.queryIdempotent(first[a], first[b])];
27    }
28 };

```

## 2.6 Max-Flow

### 2.6.1 Capacity Scaling

Gut bei dünn besetzten Graphen.

```

1 // Ford Fulkerson mit Capacity Scaling. Laufzeit: O(|E|^2*log(C))
2 static const int MAX_N = 500; // #Knoten, egal für die Laufzeit.
3 struct edge { int dest, rev; ll cap, flow; };
4 vector<edge> adjlist[MAX_N];
5 int visited[MAX_N], target, dfsCounter;
6 ll capacity;
7
8 bool dfs(int x) {
9     if (x == target) return 1;
10    if (visited[x] == dfsCounter) return 0;
11    visited[x] = dfsCounter;
12    for (edge &e : adjlist[x]) {
13        if (e.cap >= capacity && dfs(e.dest)) {
14            e.cap -= capacity; adjlist[e.dest][e.rev].cap += capacity;
15            e.flow += capacity; adjlist[e.dest][e.rev].flow -= capacity;
16            return 1;
17        }
18    }
19    return 0;
20 }
21
22 void addEdge(int u, int v, ll c) {
23     adjlist[u].push_back(edge {v, (int)adjlist[v].size(), c, 0});
24     adjlist[v].push_back(edge {u, (int)adjlist[u].size() - 1, 0, 0});
25 }
26
27 ll maxFlow(int s, int t) {
28     capacity = 1L << 62;
29     target = t;
30     ll flow = 0L;
31     while (capacity) {
32         while (dfsCounter++, dfs(s)) flow += capacity;
33         capacity /= 2;
34     }
35     return flow;

```

### 2.6.2 Dinic's Algorithm mit Capacity Scaling

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

```

1 // Laufzeit: O(|V|^2*|E|)
2 // Knoten müssen von 0 nummeriert sein.

```

```

3  const int INF = 0x3FFFFFFF, MAXN = 500;
4  struct edge { int a, b; ll f, c; };
5  int n, m, pt[MAXN], d[MAXN], s, t;
6  vector<edge> e;
7  vector<int> g[MAXN];
8  ll flow = 0, lim;
9  queue<int> q;
10
11 void addEdge(int a, int b, ll c) {
12     g[a].push_back(e.size());
13     e.push_back(edge {a, b, 0, c});
14     g[b].push_back(e.size());
15     e.push_back(edge {b, a, 0, 0});
16 }
17
18 bool bfs() {
19     for (int i = 0; i < n; i++) d[i] = INF;
20     d[s] = 0;
21     q.push(s);
22     while (!q.empty() && d[t] == INF) {
23         int cur = q.front(); q.pop();
24         for (int i = 0; i < (int)g[cur].size(); i++) {
25             int id = g[cur][i], to = e[id].b;
26             if (d[to] == INF && e[id].c - e[id].f >= lim) {
27                 d[to] = d[cur] + 1;
28                 q.push(to);
29             }
30         }
31     }
32     while (!q.empty()) q.pop();
33     return d[t] != INF;
34 }
35
36 bool dfs(int v, ll flow) {
37     if (flow == 0) return false;
38     if (v == t) return true;
39     for (; pt[v] < (int)g[v].size(); pt[v]++) {
40         int id = g[v][pt[v]], to = e[id].b;
41         if (d[to] == d[v] + 1 && e[id].c - e[id].f >= flow) {
42             int pushed = dfs(to, flow);
43             if (pushed) {
44                 e[id].f += flow;
45                 e[id ^ 1].f -= flow;
46                 return true;
47             }
48         }
49     }
50     return false;
51 }
52
53 // Nicht vergessen, s und t zu setzen!
54 void dinic() {
55     for (lim = (1LL << 62); lim >= 1; lim /= 2) {
56         if (!bfs()) continue;
57         for (int i = 0; i < n; i++) pt[i] = 0;
58         int pushed;

```

```

59         while ((pushed = dfs(s, lim))) flow += lim;
60     }
61 }

```

### 2.6.3 Anwendungen

- **Maximum Edge Disjoint Paths**

Finde die maximale Anzahl Pfade von  $s$  nach  $t$ , die keine Kante teilen.

1. Setze  $s$  als Quelle,  $t$  als Senke und die Kapazität jeder Kante auf 1.
2. Der maximale Fluss entspricht den unterschiedlichen Pfaden ohne gemeinsame Kanten.

- **Maximum Independent Paths**

Finde die maximale Anzahl an Pfaden von  $s$  nach  $t$ , die keinen Knoten teilen.

1. Setze  $s$  als Quelle,  $t$  als Senke und die Kapazität jeder Kante *und jedes Knotens* auf 1.
2. Der maximale Fluss entspricht den unterschiedlichen Pfaden ohne gemeinsame Knoten.

- **Min-Cut**

Der maximale Fluss ist gleich dem minimalen Schnitt. Bei Quelle  $s$  und Senke  $t$ , partitioniere in  $S$  und  $T$ . Zu  $S$  gehören alle Knoten, die im Residualgraphen von  $s$  aus erreichbar sind (Rückwärtskanten beachten).

### 2.7 Min-Cost-Max-Flow

```

1  static const ll flowlimit = 1LL << 60; // Größer als der maximale Fluss.
2  struct MinCostFlow { // Mit new erstellen!
3      static const int maxn = 400; // Größer als die Anzahl der Knoten.
4      static const int maxm = 5000; // Größer als die Anzahl der Kanten.
5      struct edge { int node, next; ll flow, value; } edges[maxm << 1];
6      int graph[maxn], queue[maxn], pre[maxn], con[maxn];
7      int n, m, source, target, top;
8      bool inqueue[maxn];
9      ll maxflow, mincost, dis[maxn];
10
11      MinCostFlow() { memset(graph, -1, sizeof(graph)); top = 0; }
12
13      inline int inverse(int x) { return 1 + ((x >> 1) << 2) - x; }
14
15      // Directed edge from u to v, capacity c, weight w.
16      inline int addedge(int u, int v, int c, int w) {
17          edges[top].value = w; edges[top].flow = c; edges[top].node = v;
18          edges[top].next = graph[u]; graph[u] = top++;
19          edges[top].value = -w; edges[top].flow = 0; edges[top].node = u;
20          edges[top].next = graph[v]; graph[v] = top++;
21          return top - 2;
22      }
23
24      bool SPFA() {
25          int point, node, now, head = 0, tail = 1;
26          memset(pre, -1, sizeof(pre));
27          memset(inqueue, 0, sizeof(inqueue));
28          memset(dis, 0x7F, sizeof(dis));
29          dis[source] = 0; queue[0] = source;
30          pre[source] = source; inqueue[source] = true;

```



```

31 while (head != tail) {
32     now = queue[head++];
33     point = graph[now];
34     inqueue[now] = false;
35     head %= maxn;
36
37     while (point != -1) {
38         node = edges[point].node;
39         if (edges[point].flow > 0 &&
40             dis[node] > dis[now] + edges[point].value) {
41             dis[node] = dis[now] + edges[point].value;
42             pre[node] = now; con[node] = point;
43             if (!inqueue[node]) {
44                 inqueue[node] = true; queue[tail++] = node;
45                 tail %= maxn;
46             }
47             point = edges[point].next;
48         }
49     }
50     return pre[target] != -1;
51 }
52
53 void extend() {
54     ll w = flowlimit;
55     for (int u = target; pre[u] != u; u = pre[u])
56         w = min(w, edges[con[u]].flow);
57     maxflow += w;
58     mincost += dis[target] * w;
59     for (int u = target; pre[u] != u; u = pre[u]) {
60         edges[con[u]].flow -= w;
61         edges[inverse(con[u])].flow += w;
62     }
63 }
64
65 void mincostflow() {
66     maxflow = mincost = 0;
67     while (SPFA()) extend();
68 }

```

## 2.8 Maximal Cardinality Bipartite Matching

```

1 // Laufzeit: O(n*min(ans^2, |E|))
2 // Kanten von links nach rechts. Die ersten n Knoten sind links, die anderen
3 // rechts.
4 vector<vector<int>> adjlist;
5 vector<int> pairs; // Der gematchte Knoten oder -1.
6 vector<bool> visited;
7
8 bool dfs(int v) {
9     if (visited[v]) return false;
10    visited[v] = true;
11    for (auto w : adjlist[v]) if (pairs[w] < 0 || dfs(pairs[w])) {
12        pairs[w] = v; pairs[v] = w; return true;
13    }
14    return false;
15 }

```

```

16 int kuhn(int n) { // n = #Knoten links.
17     pairs.assign(adjlist.size(), -1);
18     int ans = 0;
19     // Greedy Matching. Optionale Beschleunigung.
20     for (int i = 0; i < n; i++) for (auto w : adjlist[i])
21         if (pairs[w] == -1) { pairs[i] = w; pairs[w] = i; ans++; break; }
22     for (int i = 0; i < n; i++) if (pairs[i] == -1) {
23         visited.assign(n, false);
24         ans += dfs(i);
25     }
26     return ans; // Größe des Matchings.
27 }

```

```

1 // Laufzeit: O(sqrt(|V|)*|E|)
2 // Kanten von links nach rechts.
3 // 0: dummy Knoten, 1..n: linke Knoten, n+1..n+m: rechte Knoten
4 vector<vector<int>> adjlist;
5 vector<int> match, dist;
6
7 bool bfs(int n) {
8     queue<int> q;
9     dist[0] = INF;
10    for (int i = 1; i <= n; i++) {
11        if (match[i] == 0) { dist[i] = 0; q.push(i); }
12        else dist[i] = INF;
13    }
14    while (!q.empty()) {
15        int u = q.front(); q.pop();
16        if (dist[u] < dist[0]) for (int v : adjlist[u])
17            if (dist[match[v]] == INF) {
18                dist[match[v]] = dist[u] + 1;
19                q.push(match[v]);
20            }
21    }
22    return dist[0] != INF;
23 }
24
25 bool dfs(int u) {
26     if (u != 0) {
27         for (int v : adjlist[u])
28             if (dist[match[v]] == dist[u] + 1)
29                 if (dfs(match[v])) { match[v] = u; match[u] = v; return true; }
30         dist[u] = INF;
31         return false;
32     }
33     return true;
34 }
35
36 int hopcroft_karp(int n) { // n = #Knoten links
37     int ans = 0;
38     match.assign(adjlist.size(), 0);
39     dist.resize(adjlist.size());
40     // Greedy Matching, optionale Beschleunigung.
41     for (int i = 1; i <= n; i++) for (int w : adjlist[i])
42         if (match[w] == 0) { match[i] = w; match[w] = i; ans++; break; }
43     while (bfs(n)) for (int i = 1; i <= n; i++)

```



```

44     if(match[i] == 0 && dfs(i)) ans++;
45     return ans;
46 }

```

## 2.9 Maximum Weight Bipartite Matching

```

1 // Laufzeit: O(|V|^3)
2 int costs[N_LEFT][N_RIGHT];
3
4 // Es muss l<=r sein, ansonsten terminiert der Algorithmus nicht.
5 int match(int l, int r) {
6     vector<int> xy(l, -1), yx(r, -1), lx(l), ly(r, 0), augmenting(r);
7     vector<bool> s(l);
8     vector<ii> slack(r, ii(0,0));
9
10    for (int x = 0; x < l; x++) lx[x] = *max_element(costs[x], costs[x] + r);
11    for (int root = 0; root < l; root++) {
12        fill(augmenting.begin(), augmenting.end(), -1);
13        fill(s.begin(), s.end(), false);
14        s[root] = true;
15        for (int y = 0; y < r; y++) {
16            slack[y] = ii(lx[root] + ly[y] - costs[root][y], root);
17        }
18        int y = -1;
19        for (;;) {
20            int delta = INT_MAX, x = -1;
21            for (int yy = 0; yy < r; yy++) {
22                if (augmenting[yy] == -1) {
23                    if (slack[yy].first < delta) {
24                        delta = slack[yy].first;
25                        x = slack[yy].second;
26                        y = yy;
27                    }
28                }
29                if (delta > 0) {
30                    for (int x = 0; x < l; x++) if (s[x]) lx[x] -= delta;
31                    for (int y = 0; y < r; y++) {
32                        if (augmenting[y] > -1) ly[y] += delta;
33                        else slack[y].first -= delta;
34                    }
35                    augmenting[y] = x;
36                    x = yx[y];
37                    if (x == -1) break;
38                    s[x] = true;
39                    for (int y = 0; y < r; y++) {
40                        if (augmenting[y] == -1) {
41                            ii alt = ii(lx[x] + ly[y] - costs[x][y], x);
42                            if (slack[y].first > alt.first) {
43                                slack[y] = alt;
44                            }
45                        }
46                    }
47                    while (y != -1) {
48                        // Jede Iteration vergrößert Matching um 1 (können 0-Kanten sein!).
49                        int x = augmenting[y];
50                        int prec = xy[x];
51                        yx[y] = x;
52                        xy[x] = y;
53                        y = prec;
54                    }
55                }
56            }
57        }
58    }
59 }

```

```

52     return accumulate(lx.begin(), lx.end(), 0) +
53         accumulate(ly.begin(), ly.end(), 0); // Wert des Matchings.
54 }

```

## 2.10 Wert des maximalen Matchings

```

1 // Fehlerwahrscheinlichkeit: (n / MOD)^I
2 const int N=200, MOD=1000000007, I=10;
3 int n, adj[N][N], a[N][N];
4
5 int rank() {
6     int r = 0;
7     for (int j = 0; j < n; j++) {
8         int k = r;
9         while (k < n && !a[k][j]) ++k;
10        if (k == n) continue;
11        swap(a[r], a[k]);
12        int inv = powmod(a[r][j], MOD - 2);
13        for (int i = j; i < n; i++)
14            a[r][i] = 1LL * a[r][i] * inv % MOD;
15        for (int u = r + 1; u < n; u++)
16            for (int v = j; v < n; v++)
17                a[u][v] = (a[u][v] - 1LL * a[r][v] * a[u][j] % MOD + MOD) % MOD;
18        ++r;
19    }
20    return r;
21 }
22
23 int max_matching() {
24     int ans = 0;
25     for (int _ = 0; _ < I; _++) {
26         for (int i = 0; i < n; i++) {
27             for (int j = 0; j < i; j++) {
28                 if (adj[i][j]) {
29                     a[i][j] = rand() % (MOD - 1) + 1;
30                     a[j][i] = MOD - a[i][j];
31                 }
32             }
33             ans = max(ans, rank()/2);
34         }
35     }
36     return ans;
37 }

```

## 2.11 2-SAT

```

1 struct sat2 {
2     vector<vector<int>> adjlist, sccs;
3     vector<bool> visited, inStack;
4     int n, sccCounter, dfsCounter;
5     vector<int> d, low, idx, sol;
6     stack<int> s;
7
8     sat2(int vars) : n(vars*2) { adjlist.resize(n); };
9
10    static int var(int i) { return i << 1; }
11
12    void addImpl(int v1, int v2) {
13        adjlist[v1].push_back(v2);
14    }
15 }

```

```

14 adjlist[1^v2].push_back(1^v1);
15 }
16 void addEquiv(int v1, int v2) { addImpl(v1, v2); addImpl(v2, v1); }
17 void addOr(int v1, int v2) { addImpl(1^v1, v2); }
18 void addXor(int v1, int v2) { addOr(v1, v2); addOr(1^v1, 1^v2); }
19 void addTrue(int v1) { addImpl(1^v1, v1); }
20 void addFalse(int v1) { addTrue(1^v1); }
21 void addAnd(int v1, int v2) { addTrue(v1); addTrue(v2); }
22 void addNand(int v1, int v2) { addOr(1^v1, 1^v2); }
23
24 void dfs(int v) {
25     visited[v] = true;
26     d[v] = low[v] = dfsCounter++;
27     s.push(v); inStack[v] = true;
28
29     for (auto w : adjlist[v]) {
30         if (!visited[w]) {
31             dfs(w);
32             low[v] = min(low[v], low[w]);
33         } else if (inStack[w]) low[v] = min(low[v], low[w]);
34     }
35
36     if (d[v] == low[v]) {
37         sccs.push_back(vector<int>());
38         int w;
39         do {
40             w = s.top(); s.pop(); inStack[w] = false;
41             idx[w] = sccCounter;
42             sccs[sccCounter].push_back(w);
43         } while (w != v);
44         sccCounter++;
45     }
46
47     bool solvable() {
48         visited.assign(n, false);
49         inStack.assign(n, false);
50         d.assign(n, -1);
51         low.assign(n, -1);
52         idx.assign(n, -1);
53         sccCounter = dfsCounter = 0;
54         for (int i = 0; i < n; i++) if (!visited[i]) dfs(i);
55         for (int i = 0; i < n; i += 2) if (idx[i] == idx[i + 1]) return false;
56         return true;
57     }
58
59     void assign() {
60         sol.assign(n, -1);
61         for (int i = 0; i < sccCounter; i++) {
62             if (sol[sccs[i][0]] == -1) {
63                 for (int v : sccs[i]) {
64                     sol[v] = 1;
65                     sol[1^v] = 0;
66                 }
67             }
68         }
69     }

```

## 2.12 Bitonic TSP

```

1 // Laufzeit: O(n^2)
2 vector<vector<double>> dp, dist; // Entfernungen zwischen Punkten.
3
4 double get(int p1, int p2) {
5     int v = max(p1, p2) + 1;
6     if (v == dist.size()) return dist[p1][v - 1] + dist[p2][v - 1];
7     if (dp[p1][p2] >= 0.0) return dp[p1][p2];
8     double tryLR = dist[p1][v] + get(v, p2);
9     double tryRL = dist[p2][v] + get(p1, v);
10    return dp[p1][p2] = min(tryLR, tryRL);
11 }
12
13 void bitonicTour() {
14     dp.assign(dist.size(), vector<double>(dist.size(), -1));
15     get(0, 0); // return dp[0][0]; // Langer der Tour
16     vector<int> lr = {0}, rl = {0};
17     for (int p1 = 0, p2 = 0, v; (v = max(p1, p2) + 1) < dist.size(); ) {
18         if (dp[p1][p2] == dist[p1][v] + dp[v][p2]) {
19             lr.push_back(v); p1 = v;
20         } else {
21             rl.push_back(v); p2 = v;
22         }
23     }
24     lr.insert(lr.end(), rl.rbegin(), rl.rend()); // Tour, Knoten 0 doppelt.

```

## 3 Geometrie

### 3.1 Closest Pair

```

1 double squaredDist(pt a, pt b) {
2     return (a.fst-b.fst) * (a.fst-b.fst) + (a.snd-b.snd) * (a.snd-b.snd);
3 }
4
5 bool compY(pt a, pt b) {
6     if (a.snd == b.snd) return a.fst < b.fst;
7     return a.snd < b.snd;
8 }
9
10 // points.size() > 1 und alle Punkte mussen verschieden sein!
11 double shortestDist(vector<pt> &points) {
12     set<pt, bool(*)>(pt, pt) status(compY);
13     sort(points.begin(), points.end());
14     double opt = 1e30, sqrtOpt = 1e15;
15     auto left = points.begin(), right = points.begin();
16     status.insert(*right); right++;
17
18     while (right != points.end()) {
19         if (fabs(left->fst - right->fst) >= sqrtOpt) {
20             status.erase(*(left++));
21         } else {
22             auto lower = status.lower_bound(pt(-1e20, right->snd - sqrtOpt));
23             auto upper = status.upper_bound(pt(-1e20, right->snd + sqrtOpt));
24             while (lower != upper) {
25                 double cand = squaredDist(*right, *lower);
26                 if (cand < opt) {
27                     opt = cand;

```

```

28         sqrtOpt = sqrt(opt);
29     }
30     ++lower;
31 }
32 status.insert(*(right++));
33 }
34 return sqrtOpt;
35 }

```

### 3.2 Geraden

```

1 // Nicht complex<double> benutzen. Eigene struct schreiben.
2 struct line {
3     double a, b, c; // ax + by + c = 0; vertikale Line: b = 0, sonst: b = 1
4 };
5
6 line pointsToLine(pt p1, pt p2) {
7     line l;
8     if (fabs(p1.x - p2.x) < EPSILON) {
9         l.a = 1; l.b = 0.0; l.c = -p1.x;
10    } else {
11        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
12        l.b = 1.0;
13        l.c = -(double)(l.a * p1.x) - p1.y;
14    }
15    return l;
16 }
17
18 bool areParallel(line l1, line l2) {
19     return (fabs(l1.a - l2.a) < EPSILON) && (fabs(l1.b - l2.b) < EPSILON);
20 }
21
22 bool areSame(line l1, line l2) {
23     return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPSILON);
24 }
25
26 bool areIntersect(line l1, line l2, pt &p) {
27     if (areParallel(l1, l2)) return false;
28     p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
29     if (fabs(l1.b) > EPSILON) p.y = -(l1.a * p.x + l1.c);
30     else p.y = -(l2.a * p.x + l2.c);
31     return true;
32 }

```

### 3.3 Konvexe Hülle

```

1 // Laufzeit: O(n*log(n))
2 ll cross(const pt p, const pt a, const pt b) {
3     return (a.x - p.x) * (b.y - p.y) - (a.y - p.y) * (b.x - p.x);
4 }
5
6 // Punkte auf der konvexen Hülle, gegen den Uhrzeigersinn sortiert.
7 // Kollineare Punkte nicht enthalten, entferne dafür "=" im CCW-Test.
8 // Achtung: Der erste und letzte Punkt im Ergebnis sind gleich.
9 // Achtung: Alle Punkte müssen verschieden sein.
10 vector<pt> convexHull(vector<pt> p){
11     int n = p.size(), k = 0;

```

```

12     vector<pt> h(2 * n);
13     sort(p.begin(), p.end());
14     for (int i = 0; i < n; i++) { // Untere Hülle.
15         while (k >= 2 && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
16         h[k++] = p[i];
17     }
18     for (int i = n - 2, t = k + 1; i >= 0; i--) { // Obere Hülle.
19         while (k >= t && cross(h[k - 2], h[k - 1], p[i]) <= 0.0) k--;
20         h[k++] = p[i];
21     }
22     h.resize(k);
23     return h;
24 }

```

### 3.4 Formeln - std::complex

```

1 // Komplexe Zahlen als Darstellung für Punkte. Wenn immer möglich
2 // complex<int> verwenden. Funktionen wie abs() geben dann int zurück.
3 typedef complex<double> pt;
4
5 // Winkel zwischen Punkt und x-Achse in [0, 2 * PI).
6 double angle = arg(a);
7
8 // Punkt rotiert um Winkel theta.
9 pt a_rotated = a * exp(pt(0, theta));
10
11 // Mittelpunkt des Dreiecks abc.
12 pt centroid = (a + b + c) / 3.0;
13
14 // Skalarprodukt.
15 double dot(pt a, pt b) { return real(conj(a) * b); }
16
17 // Kreuzprodukt, 0, falls kollinear.
18 double cross(pt a, pt b) { return imag(conj(a) * b); }
19
20 // Flächeninhalt eines Dreiecks bei bekannten Eckpunkten.
21 double areaOfTriangle(pt a, pt b, pt c) {
22     return abs(cross(b - a, c - a)) / 2.0;
23 }
24
25 // Flächeninhalt eines Dreiecks bei bekannten Seitenlängen.
26 double areaOfTriangle(double a, double b, double c) {
27     double s = (a + b + c) / 2;
28     return sqrt(s * (s-a) * (s-b) * (s-c));
29 }
30
31 // Sind die Dreiecke a1, b1, c1, and a2, b2, c2 ähnlich?
32 // Erste Zeile testet Ähnlichkeit mit gleicher Orientierung,
33 // zweite Zeile testet Ähnlichkeit mit unterschiedlicher Orientierung
34 bool similar(pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
35     return (
36         (b2-a2) * (c1-a1) == (b1-a1) * (c2-a2) ||
37         (b2-a2) * (conj(c1)-conj(a1)) == (conj(b1)-conj(a1)) * (c2-a2)
38     );
39 }
40
41 // -1 => gegen den Uhrzeigersinn, 0 => kollinear, 1 => im Uhrzeigersinn.

```

```

42 // Einschränken der Rückgabe auf [-1,1] ist sicherer gegen Overflows.
43 double orientation(pt a, pt b, pt c) {
44     double orien = cross(b - a, c - a);
45     if (abs(orien) < EPSILON) return 0; // Braucht großes EPSILON: ~1e-6
46     return orien < 0 ? -1 : 1;
47 }
48
49 // Test auf Streckenschnitt zwischen a-b und c-d.
50 bool lineSegmentIntersection(pt a, pt b, pt c, pt d) {
51     if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0) {
52         double dist = abs(a - b);
53         return (abs(a - c) <= dist && abs(b - c) <= dist) ||
54             (abs(a - d) <= dist && abs(b - d) <= dist);
55     }
56     return orientation(a, b, c) * orientation(a, b, d) <= 0 &&
57         orientation(c, d, a) * orientation(c, d, b) <= 0;
58 }
59
60 // Berechnet die Schnittpunkte der Strecken a-b und c-d. Enthält entweder
61 // keinen Punkt, den einzigen Schnittpunkt oder die Endpunkte der
62 // Schnittstrecke. operator<, min, max müssen noch geschrieben werden!
63 vector<pt> lineSegmentIntersection(pt a, pt b, pt c, pt d) {
64     vector<pt> result;
65     if (orientation(a, b, c) == 0 && orientation(a, b, d) == 0 &&
66         orientation(c, d, a) == 0 && orientation(c, d, b) == 0) {
67         pt minAB = min(a, b), maxAB = max(a, b);
68         pt minCD = min(c, d), maxCD = max(c, d);
69         if (minAB < minCD && maxAB < minCD) return result;
70         if (minCD < minAB && maxCD < minAB) return result;
71         pt start = max(minAB, minCD), end = min(maxAB, maxCD);
72         result.push_back(start);
73         if (start != end) result.push_back(end);
74         return result;
75     }
76     double x1 = real(b) - real(a), y1 = imag(b) - imag(a);
77     double x2 = real(d) - real(c), y2 = imag(d) - imag(c);
78     double u1 = (-y1 * (real(a) - real(c)) + x1 * (imag(a) - imag(c))) /
79         (-x2 * y1 + x1 * y2);
80     double u2 = (x2 * (imag(a) - imag(c)) - y2 * (real(a) - real(c))) /
81         (-x2 * y1 + x1 * y2);
82     if (u1 >= 0 && u1 <= 1 && u2 >= 0 && u2 <= 1) {
83         double x = real(a) + u2 * x1, y = imag(a) + u2 * y1;
84         result.push_back(pt(x, y));
85     }
86     return result;
87 }
88
89 // Entfernung von Punkt p zur Geraden durch a-b.
90 double distToLine(pt a, pt b, pt p) {
91     return abs(cross(p - a, b - a)) / abs(b - a);
92 }
93
94 // Liegt p auf der Geraden a-b?
95 bool pointOnLine(pt a, pt b, pt p) {
96     return orientation(a, b, p) == 0;
97 }

```

```

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

```

```

154 }
155
156 // Testet, ob ein Punkt im Polygon liegt (beliebige Polygone).
157 bool pointInPolygon(pt p, vector<pt> &polygon) { // Punkte nur einmal.
158     pt rayEnd = p + pt(1, 1000000);
159     int counter = 0, n = polygon.size();
160     for (int i = 0; i < n; i++) {
161         pt start = polygon[i], end = polygon[(i + 1) % n];
162         if (lineSegmentIntersection(p, rayEnd, start, end)) counter++;
163     }
164     return counter & 1;
165 }

```

## 4 Mathe

### 4.1 ggT, kgV, erweiterter euklidischer Algorithmus

```

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

```

```

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

```

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

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

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

Falls  $d = 1$ :

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

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

```

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

```

### 4.2 Mod-Exponent über $\mathbb{F}_p$

```

1 // Laufzeit: O(log(b))
2 ll powMod(ll a, ll b, ll n) {
3     if(b == 0) return 1;
4     if(b == 1) return a % n;
5     if(b & 1) return (powMod(a, b - 1, n) * a) % n;
6     else return powMod((a * a) % n, b / 2, n);
7 }

```

Iterativ:

```

1 // Laufzeit: O(log(b))
2 ll powMod(ll a, ll b, ll n) {
3     if (b == 0) return 1;
4     ll res = 1;
5     while (b > 1) {
6         if (b & 1) res = (a * res) % n;
7         a = (a * a) % n;
8         b /= 2;
9     }
10    return (a * res) % n;
11 }

```

### 4.3 Chinesischer Restsatz

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

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

Formel kann auch für nicht teilerfremde Moduli verwendet werden. Sind die Moduli nicht teilerfremd, existiert genau dann eine Lösung, wenn  $a \equiv b \pmod{\text{ggT}(m, n)}$ . In diesem Fall sind keine Faktoren auf der linken Seite erlaubt.

```

1 // Laufzeit: O(n * log(n)), n := Anzahl der Kongruenzen
2 // Nur für teilerfremde Moduli. Berechnet das kleinste, nicht negative x,
3 // das alle Kongruenzen simultan löst. Alle Lösungen sind kongruent zum
4 // kgV der Moduli (Produkt, falls alle teilerfremd sind).
5 struct ChineseRemainder {
6     typedef __int128 lll;
7     vector<lll> lhs, rhs, mods, inv;
8     lll M; // Produkt über die Moduli. Kann leicht überlaufen.
9
10    ll g(vector<lll> &vec) {
11        lll res = 0;
12        for (int i = 0; i < (int)vec.size(); i++) {
13            res += (vec[i] * inv[i]) % M;
14            res %= M;
15        }
16        return res;
17    }
18
19    // Fügt Kongruenz l * x = r (mod m) hinzu.
20    void addEquation(ll l, ll r, ll m) {
21        lhs.push_back(l);

```

```

22     rhs.push_back(r);
23     mods.push_back(m);
24 }
25
26 // Löst das System.
27 ll solve() {
28     M = accumulate(mods.begin(), mods.end(), 1ll(1), multiplies<ll>());
29     inv.resize(lhs.size());
30     for (int i = 0; i < (int)lhs.size(); i++) {
31         ll x = (M / mods[i]) % mods[i];
32         inv[i] = (multInv(x, mods[i]) * (M / mods[i]));
33     }
34     return (multInv(g(lhs), M) * g(rhs)) % M;
35 }
36 };

```

#### 4.4 Primzahltest & Faktorisierung

```

1 bool isPrime(ll n) { // Miller Rabin Primzahltest. O(log n)
2     if(n == 2) return true;
3     if(n < 2 || n % 2 == 0) return false;
4     ll d = n - 1, j = 0;
5     while(d % 2 == 0) d >>= 1, j++;
6     for(int a = 2; a <= min((ll)37, n - 1); a++) {
7         ll v = powMod(a, d, n); // Implementierung von oben.
8         if(v == 1 || v == n - 1) continue;
9         for(int i = 1; i <= j; i++) {
10             v = (v * v) % n;
11             if(v == n - 1 || v <= 1) break;
12         }
13         if(v != n - 1) return false;
14     }
15     return true;
16 }
17
18 ll rho(ll n) { // Findet Faktor < n, nicht unbedingt prim.
19     if (~n & 1) return 2;
20     ll c = rand() % n, x = rand() % n, y = x, d = 1;
21     while (d == 1) {
22         x = ((x * x) % n + c) % n;
23         y = ((y * y) % n + c) % n;
24         y = ((y * y) % n + c) % n;
25         d = gcd(abs(x - y), n); // Implementierung von oben.
26     }
27     return d == n ? rho(n) : d;
28 }
29
30 void factor(ll n, map<ll, int> &facts) {
31     if (n == 1) return;
32     if (isPrime(n)) {
33         facts[n]++;
34         return;
35     }
36     ll f = rho(n);
37     factor(n / f, facts);
38     factor(f, facts);
39 }

```

#### 4.5 Primzialsieb von ERATOSTHENES

```

1 // Laufzeit: O(n * log log n)
2 // Kann erweitert werden: Für jede Zahl den kleinsten Primfaktor.
3 // Dabei vorsicht: Nicht kleinere Faktoren überschreiben.
4 #define N 100000000 // Bis 10^8 in unter 64MB Speicher.
5 bitset<N / 2> isNotPrime;
6
7 inline bool isPrime(int x) { // Diese Methode zum Lookup verwenden.
8     if (x < 2) return false;
9     else if (x == 2) return true;
10    else if (!(x & 1)) return false;
11    else return !isNotPrime[x / 2];
12 }
13
14 inline int primeSieve() { // Rückgabe: Anzahl der Primzahlen < N.
15     int counter = 1; // Die 2, die sonst vergessen würde.
16     for (int i = 3; i < N; i += 2) {
17         if (!isNotPrime[i / 2]) {
18             for (int j = i * i; j < N; j += 2 * i) isNotPrime[j / 2] = 1;
19             counter++;
20         }
21     }
22     return counter;

```

#### 4.6 EULERSCHE $\varphi$ -FUNKTION

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

```

1 ll phi(ll n) { // Laufzeit: O(sqrt(n))
2     // Optimierung: Falls n prim, n - 1 zurückgeben (Miller-Rabin/Sieb).
3     ll result = n;
4     for(int i = 2; i * i <= n; ++i) {
5         if(n % i == 0) { // Optimierung: Nur über Primzahlen iterieren.
6             while(n % i == 0) n /= i;
7             result -= result / i;
8         }
9     }
10    if(n > 1) result -= result / n;
11    return result;
12 }
13
14 // Sieb, falls alle Werte benötigt werden. Laufzeit: O(N*log(log(N)))
15 for (int i = 1; i <= N; i++) phi[i] = i;
16 for (int i = 2; i <= N; i++) if (phi[i] == i) {
17     for (int j = i; j <= N; j += i) {
18         phi[j] /= i;
19         phi[j] *= i - 1;

```

20 } }

#### 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  $g^k \equiv 1 \pmod n$ , ist  $k = \varphi(n)$ .

```

1 // Ist g Primitivwurzel modulo p. Teste zufällige g, um eine zu finden.
2 bool is_primitive(ll g, ll p) {
3     map<ll, int> facs;
4     factor(p - 1, facs);
5     for (auto &f : facs)
6         if (1 == powMod(g, (p - 1) / f.first, p)) return false;
7     return true;
8 }
9
10 // Alternativ: Generator zum Finden. -1 falls keine existiert.
11 ll generator (ll p) { // Laufzeit: O(ans*log(phi(n))*log(n))
12     map<ll, int> facs;
13     factor(n, facs);
14     ll phi = phi(p), n = phi;
15
16     for (ll res = 2; res <= p; res++) {
17         bool ok = true;
18         for (auto &f : facs)
19             ok &= powMod(res, phi / f.first, p) != 1;
20         if (ok) return res;
21     }
22     return -1;
23 }

```

#### 4.8 Diskreter Logarithmus

```

1 // Bestimmt Lösung x für a^x=b mod m.
2 ll solve (ll a, ll b, ll m) { // Laufzeit: O(sqrt(m)*log(m))
3     ll n = (ll)sqrt((double)m) + 1;
4     map<ll, ll> vals;
5     for (int i = n; i >= 1; i--) vals[powMod(a, i * n, m)] = i;
6     for (int i = 0; i <= n; i++) {
7         ll cur = (powMod(a, i, m) * b) % m;
8         if (vals.count(cur)) {
9             ll ans = vals[cur] * n - i;
10            if (ans < m) return ans;
11        }
12    }
13    return -1;
14 }

```

#### 4.9 Binomialkoeffizienten

```

1 // Laufzeit: O(k)
2 ll calc_binom(ll n, ll k) { // Sehr sicher gegen Overflows.
3     ll r = 1, d;

```

```

4     if (k > n) return 0;
5     for (d = 1; d <= k; d++) { // Reihenfolge garantiert Teilbarkeit.
6         r *= n--;
7         r /= d;
8     }
9     return r;
10 }

```

#### 4.10 LGS über $\mathbb{F}_p$

```

1 // Laufzeit: O(n^3)
2 void swapLines(int n, int l1, int l2) {
3     for (int i = 0; i <= n; i++) swap(mat[l1][i], mat[l2][i]);
4 }
5
6 void normalLine(int n, int line, ll p) {
7     ll factor = multInv(mat[line][line], p); // Implementierung von oben.
8     for (int i = 0; i <= n; i++) {
9         mat[line][i] *= factor;
10        mat[line][i] %= p;
11    }
12 }
13 void takeAll(int n, int line, ll p) {
14     for (int i = 0; i < n; i++) {
15         if (i == line) continue;
16         ll diff = mat[i][line];
17         for (int j = 0; j <= n; j++) {
18             mat[i][j] -= (diff * mat[line][j]) % p;
19             mat[i][j] %= p;
20             if (mat[i][j] < 0) mat[i][j] += p;
21         }
22     }
23 }
24 void gauss(int n, ll p) { // nx(n+1)-Matrix, Körper F_p.
25     for (int line = 0; line < n; line++) {
26         int swappee = line;
27         while (mat[swappee][line] == 0) swappee++;
28         swapLines(n, line, swappee);
29         normalLine(n, line, p);
30         takeAll(n, line, p);
31     }
32 }

```

#### 4.11 LGS über $\mathbb{R}$

```

1 // Laufzeit: O(n^3)
2 void swapLines(int n, int l1, int l2) {
3     for (int i = 0; i <= n; i++) swap(mat[l1][i], mat[l2][i]);
4 }
5
6 void normalLine(int n, int line) {
7     double factor = mat[line][line];
8     for (int i = 0; i <= n; i++) {
9         mat[line][i] /= factor;
10    }
11 }
12 void takeAll(int n, int line) {
13     for (int i = 0; i < n; i++) {
14         if (i == line) continue;

```



```

15     double diff = mat[i][line];
16     for (int j = 0; j <= n; j++) {
17         mat[i][j] -= diff * mat[line][j];
18     }}
19
20 int gauss(int n) { // Gibt zurück, ob das System (eindeutig) lösbar ist.
21     vector<bool> done(n, false);
22     for (int i = 0; i < n; i++) {
23         int swappee = i; // Sucht Pivotzeile für bessere Stabilität.
24         for (int j = 0; j < n; j++) {
25             if (done[j]) continue;
26             if (abs(mat[j][i]) > abs(mat[i][i])) swappee = j;
27         }
28         swapLines(n, i, swappee);
29         if (abs(mat[i][i]) > EPSILON) {
30             normalLine(n, i);
31             takeAll(n, i);
32             done[i] = true;
33         }} // Ab jetzt nur noch checks bzgl. Eindeutigkeit/Existenz der Lösung.
34     for (int i = 0; i < n; i++) {
35         bool allZero = true;
36         for (int j = i; j < n; j++)
37             if (abs(mat[i][j]) > EPSILON) allZero = false;
38         if (allZero && abs(mat[i][n]) > EPSILON) return INCONSISTENT;
39         if (allZero && abs(mat[i][n]) < EPSILON) return MULTIPLE;
40     }
41     return UNIQUE;
42 }

```

#### 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]])`

```

1 // Laufzeit:  $O(n \log(n))$ .
2 typedef complex<double> cplx; // Eigene Implementierung ist schneller.
3
4 // a.size() muss eine Zweierpotenz sein!
5 vector<cplx> fft(const vector<cplx> &a, bool inverse = 0) {
6     int logn = 1, n = a.size();
7     vector<cplx> A(n);
8     while ((1 << logn) < n) logn++;
9     for (int i = 0; i < n; i++) {
10         int j = 0;
11         for (int k = 0; k < logn; k++) j = (j << 1) | ((i >> k) & 1);
12         A[j] = a[i];
13     }
14     for (int s = 2; s <= n; s <= 1) {
15         double angle = 2 * PI / s * (inverse ? -1 : 1);
16         cplx ws(cos(angle), sin(angle));
17         for (int j = 0; j < n; j += s) {
18             cplx w = 1;
19             for (int k = 0; k < s / 2; k++) {
20                 cplx u = A[j + k], t = A[j + s / 2 + k];

```

```

21         A[j + k] = u + w * t;
22         A[j + s / 2 + k] = u - w * t;
23         if (inverse) A[j + k] /= 2, A[j + s / 2 + k] /= 2;
24         w *= ws;
25     }}
26     return A;
27 }
28
29 // Polynome: a[0] = a_0, a[1] = a_1, ... und b[0] = b_0, b[1] = b_1, ...
30 // Bei Integeren: Runde Koeffizienten: (int)round(a[i].real())
31 vector<cplx> a = {0,0,0,0,1,2,3,4}, b = {0,0,0,0,2,3,0,1};
32 a = fft(a); b = fft(b);
33 for (int i = 0; i < (int)a.size(); i++) a[i] *= b[i];
34 a = fft(a,1); // a = a * b

```

#### 4.13 Numerisch Integrieren, Simpsonregel

```

1 double f(double x) { return x; }
2
3 double simps(double a, double b) {
4     return (f(a) + 4.0 * f((a + b) / 2.0) + f(b)) * (b - a) / 6.0;
5 }
6
7 double integrate(double a, double b) {
8     double m = (a + b) / 2.0;
9     double l = simps(a, m), r = simps(m, b), tot = simps(a, b);
10    if (abs(l + r - tot) < EPSILON) return tot;
11    return integrate(a, m) + integrate(m, b);
12 }

```

#### 4.14 3D-Kugeln

```

1 // Great Circle Distance mit Längen- und Breitengrad.
2 double gcDist(
3     double pLat, double pLon, double qLat, double qLon, double radius) {
4     pLat *= PI / 180; pLon *= PI / 180; qLat *= PI / 180; qLon *= PI / 180;
5     return radius * acos(cos(pLat) * cos(pLon) * cos(qLat) * cos(qLon) +
6         cos(pLat) * sin(pLon) * cos(qLat) * sin(qLon) +
7         sin(pLat) * sin(qLat));
8 }
9
10 // Great Circle Distance mit kartesischen Koordinaten.
11 double gcDist(point p, point q) {
12     return acos(p.x * q.x + p.y * q.y + p.z * q.z);
13 }
14
15 // 3D Punkt in kartesischen Koordinaten.
16 struct point {
17     double x, y, z;
18     point() {}
19     point(double x, double y, double z) : x(x), y(y), z(z) {}
20     point(double lat, double lon) {
21         lat *= PI / 180.0; lon *= PI / 180.0;
22         x = cos(lat) * sin(lon); y = cos(lat) * cos(lon); z = sin(lat);
23     }
24 };

```

#### 4.15 Longest Increasing Subsequence

```

1 vector<int> longestIncreasingSubsequence(vector<int> &seq) {
2     int n = seq.size(), lisLength = 0, lisEnd = 0;
3     vector<int> L(n), L_id(n), parents(n);
4     for (int i = 0; i < n; i++) {
5         int pos =
6             lower_bound(L.begin(), L.begin() + lisLength, seq[i]) - L.begin();
7         L[pos] = seq[i];
8         L_id[pos] = i;
9         parents[i] = pos ? L_id[pos - 1] : -1;
10        if (pos + 1 > lisLength) {
11            lisLength = pos + 1;
12            lisEnd = i;
13        }
14        // Ab hier Rekonstruktion der Sequenz.
15        vector<int> result(lisLength);
16        int pos = lisLength - 1, x = lisEnd;
17        while (parents[x] >= 0) {
18            result[pos--] = x;
19            x = parents[x];
20        }
21        result[0] = x;
22        return result; // Liste mit Indizes einer LIS.
23    }

```

#### 4.16 Inversionszahl und Mergesort

```

1 // Laufzeit: O(n*log(n))
2 ll merge(vector<ll> &v, vector<ll> &left, vector<ll> &right) {
3     int a = 0, b = 0, i = 0;
4     ll inv = 0;
5     while (a < (int)left.size() && b < (int)right.size()) {
6         if (left[a] < right[b]) v[i++] = left[a++];
7         else {
8             inv += left.size() - a;
9             v[i++] = right[b++];
10        }
11    }
12    while (a < (int)left.size()) v[i++] = left[a++];
13    while (b < (int)right.size()) v[i++] = right[b++];
14    return inv;
15 }
16
17 ll mergeSort(vector<ll> &v) { // Sortiert v und gibt Inversionszahl zurück.
18     int n = v.size();
19     vector<ll> left(n / 2), right((n + 1) / 2);
20     for (int i = 0; i < n / 2; i++) left[i] = v[i];
21     for (int i = n / 2; i < n; i++) right[i - n / 2] = v[i];
22
23     ll result = 0;
24     if (left.size() > 1) result += mergeSort(left);
25     if (right.size() > 1) result += mergeSort(right);
26     return result + merge(v, left, right);
27 }

```

#### 4.17 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  $X_1, \dots, X_k$  hat, dann ist die GRUNDY-Zahl des Gesamtzustandes  $g(X_1) \oplus \dots \oplus g(X_k)$ .

#### 4.18 LEGENDRE-Symbol

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

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

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

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

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

```

1 int legendre(ll a, ll p) {
2     a %= p;
3     if (a == 0) return 0;
4     if (a == 1 || p == 2) return 1;
5     if (a == 2) return (((p * p - 1) / 8) & 1) ? -1 : 1;
6     if (isPrime(a)) {
7         return legendre(p, a) * (((p - 1) * (a - 1) / 4) & 1) ? -1 : 1;
8     } else {
9         map<ll, int> facts;
10        factor(a, facts);
11        int res = 1;
12        for (auto f : facts)
13            if (f.second & 1)
14                res *= legendre(f.first, p);
15        return res;
16    }
17 }

```

#### 4.19 Möbius-Funktion und Möbius-Inversion

- Seien  $f, g : \mathbb{N} \rightarrow \mathbb{N}$  und  $g(n) := \sum_{d \mid n} f(d)$ . Dann ist  $f(n) = \sum_{d \mid n} g(d) \mu\left(\frac{n}{d}\right)$ .
- $\sum_{d \mid n} \mu(d) = \begin{cases} 1 & \text{falls } n = 1 \\ 0 & \text{sonst} \end{cases}$

**Beispiel Inklusion/Exklusion:** Gegeben sein eine Sequenz  $A = a_1, \dots, a_n$  von Zahlen,  $1 \leq a_i \leq N$ . Zähle die Anzahl der *coprime subsequences*.

**Lösung:** Für jedes  $x$ , sei  $\text{cnt}[x]$  die Anzahl der Vielfachen von  $x$  in  $A$ . Es gibt  $2^{\text{cnt}[x]} - 1$  nicht leere Subsequences in  $A$ , die nur Vielfache von  $x$  enthalten. Die Anzahl der Subsequences mit  $\text{ggT} = 1$  ist gegeben durch  $\sum_{i=1}^N \mu(i) \cdot (2^{\text{cnt}[i]} - 1)$ .

```

1 // Laufzeit: O(N*log(log(N)))
2 int mu[N+1]; mu[1] = 1;
3 for (int i = 1; i <= N; i++) {
4     for (int j = 2 * i; j <= N; j += i) mu[j] -= mu[i];

```

## 4.20 Kombinatorik

Berühmte Zahlen	
FIBONACCI	$f(0) = 0 \quad f(1) = 1 \quad f(n+2) = f(n+1) + f(n)$
CATALAN	$C_0 = 1 \quad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1} \binom{2n}{n} = \frac{2(2n-1)}{n+1} \cdot C_{n-1}$
EULER I	$\langle n \rangle = \langle n-1 \rangle = 1 \quad \langle n \rangle_k = (k+1) \langle n-1 \rangle_k + (n-k) \langle n-1 \rangle_{k-1}$
EULER II	$\langle\langle n \rangle\rangle = 1 \quad \langle\langle n \rangle\rangle_k = 0 \quad \langle\langle n \rangle\rangle_k = (k+1) \langle\langle n-1 \rangle\rangle_k + (2n-k-1) \langle\langle n-1 \rangle\rangle_{k-1}$
STIRLING I	$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1 \quad \begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0 \quad \begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix}$
STIRLING II	$\begin{Bmatrix} n \\ 1 \end{Bmatrix} = \begin{Bmatrix} n \\ n \end{Bmatrix} = 1 \quad \begin{Bmatrix} n \\ k \end{Bmatrix} = k \begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}$
BELL	$B_1 = 1 \quad B_n = \sum_{k=0}^{n-1} B_k \binom{n-1}{k} = \sum_{k=0}^n \begin{Bmatrix} n \\ k \end{Bmatrix}$
PARTITIONS	$f(0,0) = 1 \quad f(n,k) = 0 \text{ für } k > n \text{ oder } n \leq 0 \text{ oder } k \leq 0$ $f(n,k) = f(n-k,k) + f(n-1,k-1)$

**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  $\{1, \dots, n\}$  mit genau  $k$  Anstiegen. Für die  $n$ -te Zahl gibt es  $n$  mögliche Positionen zum Einfügen. Dabei wird entweder ein Anstieg in zwei gesplittet oder ein Anstieg um  $n$  ergänzt.

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

**STIRLING-Zahlen 1. Ordnung** Die Anzahl der Permutationen von  $\{1, \dots, n\}$  mit genau  $k$  Zyklen. Es gibt zwei Möglichkeiten für die  $n$ -te Zahl. Entweder sie bildet einen eigenen 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 Teilmen-gen 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  $\{1, \dots, n\}$ . Wie STIRLING-Zahlen 2. Ordnung ohne Limit durch  $k$ .

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

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

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

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

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

BERTRAND'S Ballot Theorem (Kandidaten $A$ und $B$ , $k \in \mathbb{N}$ )			
$\#A > k\#B$	$Pr = \frac{a-kb}{a+b}$	$\#B - \#A \leq k$	$Pr = 1 - \frac{ab!}{(a+k+1)!(b-k-1)!}$
$\#A \geq k\#B$	$Pr = \frac{a+1-kb}{a+1}$	$\#A \geq \#B + k$	$Num = \frac{a-k+1-b}{a-k+1} \binom{a+b-k}{b}$

The Twelvelfold Way (verteile $n$ Bälle auf $k$ Boxen)				
Bälle Boxen	identisch identisch	unterscheidbar identisch	identisch unterscheidbar	unterscheidbar unterscheidbar
-	$p_k(n)$	$\sum_{i=0}^k \binom{n}{i}$	$\binom{n+k-1}{k-1}$	$k^n$
size $\geq 1$	$p(n,k)$	$\binom{n}{k}$	$\binom{n-1}{k-1}$	$k! \binom{n}{k}$
size $\leq 1$	$[n \leq k]$	$[n \leq k]$	$\binom{k}{n}$	$n! \binom{k}{n}$
$p_k(n)$ : #Anzahl der Partitionen von $n$ in $\leq k$ positive Summanden. $p(n,k)$ : #Anzahl der Partitionen von $n$ in genau $k$ positive Summanden. [Bedingung]: <b>return</b> 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)			$(n^{30} + 15n^{16} + 20n^{10} + 24n^6)/60$	

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 $n \equiv a \pmod{a+1}$ $SG_n = n \% (a+1) \% 2$ , sonst.
$M_{\text{❶}} = \left\lfloor \frac{pile_i}{2} \right\rfloor$ $M_{\text{❷}} = \left\lceil \frac{pile_i}{2} \right\rceil, pile_i\}$	❶ $SG_{2n} = n, SG_{2n+1} = SG_n$ ❷ $SG_0 = 0, SG_n = \lceil \log_2 n \rceil + 1$
$M_{\text{❶}} = \text{Teiler von } pile_i$ $M_{\text{❷}} = \text{echte Teiler von } pile_i$	❶ $SG_0 = 0, SG_n = SG_{\text{②},n} + 1$ ❷ $ST_1 = 0, SG_n = \# \text{Nullen am Ende von } n_{bin}$
$M_{\text{❶}} = [k]$ $M_{\text{❷}} = S, (S \text{ endlich})$ $M_{\text{❸}} = S \cup \{pile_i\}$	$SG_{\text{❶},n} = n \pmod{k+1}$ ❶ Niederlage bei $SG = 0$ ❷ Niederlage bei $SG = 1$ $SG_{\text{❸},n} = SG_{\text{❷},n} + 1$
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 $n \equiv 1 \pmod{k+1}$ . 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 $n \equiv 1, 2 \pmod{4}$ $SG_n = n+1$ , falls $n \equiv 3 \pmod{4}$ $SG_n = n-1$ , falls $n \equiv 0 \pmod{4}$
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.

## Verschiedenes

Türme von Hanoi, minimale Schrittzahl:	$T_n = 2^n - 1$
#Regionen zwischen $n$ Gearden	$\frac{n(n+1)}{2} + 1$
#geschlossene 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	$\frac{k}{n} \binom{n}{k} n^{n-k}$
Derangements	$!n = (n-1)!(n-1) + (n-2)!$ $!n = \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor$ $\lim_{n \rightarrow \infty} \frac{!n}{n!} = \frac{1}{e}$

## 5 Strings

## 5.1 KNUTH-MORRIS-PRATT-Algorithmus

```

1 // Laufzeit: O(n + m), n = #Text, m = #Pattern
2 vector<int> kmpPreprocessing(string &sub) {
3     vector<int> b(sub.length() + 1);
4     b[0] = -1;
5     int i = 0, j = -1;
6     while (i < (int)sub.length()) {
7         while (j >= 0 && sub[i] != sub[j]) j = b[j];
8         i++; j++;
9         b[i] = j;
10    }
11    return b;
12 }
13
14 vector<int> kmpSearch(string &s, string &sub) {
15     vector<int> pre = kmpPreprocessing(sub), result;
16     int i = 0, j = 0;
17     while (i < (int)s.length()) {
18         while (j >= 0 && s[i] != sub[j]) j = pre[j];
19         i++; j++;
20         if (j == (int)sub.length()) {
21             result.push_back(i - j);
22             j = pre[j];
23         }
24     }
25     return result;
26 }

```

## 5.2 AHO-CORASICK-Automat

```

1 // Laufzeit: O(n + m + z), n = #Text, m = Summe #Pattern, z = #Matches
2 // Findet mehrere Patterns gleichzeitig in einem String.
3 // 1) Wurzel erstellen: aho.push_back(vertex());
4 // 2) Mit addString(0, pattern, idx); Patterns hinzufügen.
5 // 3) finishAutomaton(0) aufrufen.
6 // 4) Mit state = go(state, c) in nächsten Zustand wechseln.
7 // DANACH: Wenn patterns-Vektor nicht leer ist: Hier enden alle
8 // enthaltenen Patterns.
9 // ACHTUNG: Die Zahlenwerte der auftretenden Buchstaben müssen
10 // zusammenhängend sein und bei 0 beginnen!

```

```

11 struct vertex {
12     int next[ALPHABET_SIZE], failure;
13     int character;
14     vector<int> patterns; // Indizes der Patterns, die hier enden.
15     vertex() { for (int i = 0; i < ALPHABET_SIZE; i++) next[i] = -1; }
16 };
17 vector<vertex> aho;
18
19 void addString(int v, vector<int> &pattern, int patternIdx) {
20     for (int i = 0; i < (int)pattern.size(); i++) {
21         if (aho[v].next[pattern[i]] == -1) {
22             aho[v].next[pattern[i]] = aho.size();
23             aho.push_back(vertex());
24             aho.back().character = pattern[i];
25         }
26         v = aho[v].next[pattern[i]];
27     }
28     aho[v].patterns.push_back(patternIdx);
29 }
30
31 void finishAutomaton(int v) {
32     for (int i = 0; i < ALPHABET_SIZE; i++)
33         if (aho[v].next[i] == -1) aho[v].next[i] = v;
34
35     queue<int> q;
36     for (int i = 0; i < ALPHABET_SIZE; i++) {
37         if (aho[v].next[i] != v) {
38             aho[aho[v].next[i]].failure = v;
39             q.push(aho[v].next[i]);
40         }
41     }
42     while (!q.empty()) {
43         int r = q.front(); q.pop();
44         for (int i = 0; i < ALPHABET_SIZE; i++) {
45             if (aho[r].next[i] != -1) {
46                 q.push(aho[r].next[i]);
47                 int f = aho[r].failure;
48                 while (aho[f].next[i] == -1) f = aho[f].failure;
49                 aho[aho[r].next[i]].failure = aho[f].next[i];
50                 for (int j = 0; j < (int)aho[aho[f].next[i]].patterns.size(); j++) {
51                     aho[aho[r].next[i]].patterns.push_back(
52                         aho[aho[f].next[i]].patterns[j]);
53                 }
54             }
55         }
56     }
57
58     int go(int v, int c) {
59         if (aho[v].next[c] != -1) return aho[v].next[c];
60         else return go(aho[v].failure, c);
61     }
62 }

```

## 5.3 Trie

```

1 // Zahlenwerte müssen bei 0 beginnen und zusammenhängend sein.
2 struct node {
3     int children[ALPHABET_SIZE], c; // c = #Wörter, die hier enden.
4     node() {
5         idx = -1;
6         for (int i = 0; i < ALPHABET_SIZE; i++) children[i] = -1;
7     }
8 }

```

```

8 };
9 vector<node> trie; // Anlegen mit trie.push_back(node());
10
11 void insert(int vert, vector<int> &txt, int s) { // Laufzeit: O(|txt|)
12     if (s == (int)txt.size()) { trie[vert].c++; return; }
13     if (trie[vert].children[txt[s]] == -1) {
14         trie[vert].children[txt[s]] = trie.size();
15         trie.push_back(node());
16     }
17     insert(trie[vert].children[txt[s]], txt, s + 1);
18 }
19
20 int contains(int vert, vector<int> &txt, int s) { // Laufzeit: O(|txt|)
21     if (s == (int)txt.size()) return trie[vert].c;
22     if (trie[vert].children[txt[s]] != -1) {
23         return contains(trie[vert].children[txt[s]], txt, s + 1);
24     } else return 0;
25 }

```

## 5.4 Suffix-Baum

```

1 // Baut Suffixbaum online auf. Laufzeit: O(n)
2 // Einmal initSuffixTree() aufrufen und dann extend für jeden Buchstaben.
3 // '\0'-Zeichen (oder ähnliches) an den Text anhängen!
4 string s;
5 int root, lastIdx, needsSuffix, pos, remainder, curVert, curEdge, curLen;
6 struct Vert {
7     int start, end, suffix; // Kante [start,end)
8     map<char, int> next;
9     int len() { return min(end, pos + 1) - start; }
10 };
11 vector<Vert> tree;
12
13 int newVert(int start, int end) {
14     Vert v;
15     v.start = start;
16     v.end = end;
17     v.suffix = 0;
18     tree.push_back(v);
19     return ++lastIdx;
20 }
21
22 void addSuffixLink(int vert) {
23     if (needsSuffix) tree[needsSuffix].suffix = vert;
24     needsSuffix = vert;
25 }
26
27 bool fullImplicitEdge(int vert) {
28     if (curLen >= tree[vert].len()) {
29         curEdge += tree[vert].len();
30         curLen -= tree[vert].len();
31         curVert = vert;
32         return true;
33     }
34     return false;
35 }
36

```

```

37 void initSuffixTree() {
38     needsSuffix = remainder = curEdge = curLen = 0;
39     lastIdx = pos = -1;
40     root = curVert = newVert(-1, -1);
41 }
42
43 void extend() {
44     pos++;
45     needsSuffix = 0;
46     remainder++;
47     while (remainder) {
48         if (curLen == 0) curEdge = pos;
49         if (!tree[curVert].next.count(s[curEdge])) {
50             int leaf = newVert(pos, s.size());
51             tree[curVert].next[s[curEdge]] = leaf;
52             tree[curVert].next[s[curEdge]] = leaf;
53             addSuffixLink(curVert);
54         } else {
55             int nxt = tree[curVert].next[s[curEdge]];
56             if (fullImplicitEdge(nxt)) continue;
57             if (s[tree[nxt].start + curLen] == s[pos]) {
58                 curLen++;
59                 addSuffixLink(curVert);
60                 break;
61             }
62             int split = newVert(tree[nxt].start, tree[nxt].start + curLen);
63             tree[curVert].next[s[curEdge]] = split;
64             int leaf = newVert(pos, s.size());
65             tree[split].next[s[pos]] = leaf;
66             tree[nxt].start += curLen;
67             tree[split].next[s[tree[nxt].start]] = nxt;
68             addSuffixLink(split);
69         }
70         remainder--;
71         if (curVert == root && curLen) {
72             curLen--;
73             curEdge = pos - remainder + 1;
74         } else {
75             curVert = tree[curVert].suffix ? tree[curVert].suffix : root;
76         }
77     }
78 }

```

## 5.5 Suffix-Array

```

1 struct SuffixArray { // MAX_LG = ceil(log2(MAX_N))
2     static const int MAX_N = 100010, MAX_LG = 17;
3     pair<pair<int, int>, int> L[MAX_N];
4     int P[MAX_LG + 1][MAX_N], n, step, count;
5     int suffixArray[MAX_N], lcpArray[MAX_N];
6
7     SuffixArray(const string &s) : n(s.size()) { // Laufzeit: O(n*log^2(n))
8         for (int i = 0; i < n; i++) P[0][i] = s[i];
9         suffixArray[0] = 0; // Falls n == 1.
10        for (step = 1, count = 1; count < n; step++, count <= 1) {
11            for (int i = 0; i < n; i++) L[i] =
12                {{P[step-1][i], i+count < n ? P[step-1][i+count] : -1}, i};

```



```

13     sort(L, L + n);
14     for (int i = 0; i < n; i++) P[step][L[i].second] = i > 0 &&
15         L[i].first == L[i-1].first ? P[step][L[i-1].second] : i;
16 }
17 for (int i = 0; i < n; i++) suffixArray[i] = L[i].second;
18 for (int i = 1; i < n; i++)
19     lcpArray[i] = lcp(suffixArray[i - 1], suffixArray[i]);
20 }
21
22 // x und y sind Indizes im String, nicht im Suffixarray.
23 int lcp(int x, int y) { // Laufzeit: O(log(n))
24     int k, ret = 0;
25     if (x == y) return n - x;
26     for (k = step - 1; k >= 0 && x < n && y < n; k--)
27         if (P[k][x] == P[k][y])
28             x += 1 << k, y += 1 << k, ret += 1 << k;
29     return ret;
30 }
31 };

```

## 5.6 Suffix-Automaton

```

1 #define ALPHABET_SIZE 26
2 struct SuffixAutomaton {
3     struct State {
4         int length; int link; int next[ALPHABET_SIZE];
5         State() { memset(next, 0, sizeof(next)); }
6     };
7     static const int MAX_N = 100000; // Maximale Länge des Strings.
8     State states[2 * MAX_N];
9     int size, last;
10
11     SuffixAutomaton(string &s) { // Laufzeit: O(|s|)
12         size = 1; last = 0;
13         states[0].length = 0;
14         states[0].link = -1;
15         for (auto c : s) extend(c);
16     }
17
18     void extend(char c) {
19         c -= 'a'; // Werte von c müssen bei 0 beginnen.
20         int current = size++;
21         states[current].length = states[last].length + 1;
22         int pos = last;
23         while (pos != -1 && !states[pos].next[(int)c]) {
24             states[pos].next[(int)c] = current;
25             pos = states[pos].link;
26         }
27         if (pos == -1) states[current].link = 0;
28         else {
29             int q = states[pos].next[(int)c];
30             if (states[pos].length + 1 == states[q].length) {
31                 states[current].link = q;
32             } else {
33                 int clone = size++;
34                 states[clone].length = states[pos].length + 1;
35                 states[clone].link = states[q].link;

```

```

36         memcpy(states[clone].next, states[q].next,
37             sizeof(states[q].next));
38         while (pos != -1 && states[pos].next[(int)c] == q) {
39             states[pos].next[(int)c] = clone;
40             pos = states[pos].link;
41         }
42         states[q].link = states[current].link = clone;
43     }
44     last = current;
45 }
46
47 // Paar mit Startposition und Länge des LCS. Index in Parameter s.
48 int longestCommonSubstring(string &s) { // Laufzeit: O(|s|)
49     int v = 0, l = 0, best = 0, bestpos = 0;
50     for (int i = 0; i < (int)s.size(); i++) {
51         int c = s[i] - 'a';
52         while (v && !states[v].next[c]) {
53             v = states[v].link;
54             l = states[v].length;
55         }
56         if (states[v].next[c]) { v = states[v].next[c]; l++; }
57         if (l > best) { best = l; bestpos = i; }
58     }
59     return ii(bestpos - best + 1, best);
60 }
61
62 // Berechnet die Terminale des Automaten.
63 vector<int> calculateTerminals() {
64     vector<int> terminals;
65     int pos = last;
66     while (pos != -1) {
67         terminals.push_back(pos);
68         pos = states[pos].link;
69     }
70     return terminals;
71 }
72 };

```

- **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

```

1 // Laufzeit: O(|a|*|b|)
2 string lcsc(string &a, string &b) {
3     int m[a.length() + 1][b.length() + 1], x=0, y=0;

```



```

4  memset(m, 0, sizeof(m));
5  for(int y = a.length() - 1; y >= 0; y--) {
6      for(int x = b.length() - 1; x >= 0; x--) {
7          if(a[y] == b[x]) m[y][x] = 1 + m[y+1][x+1];
8          else m[y][x] = max(m[y+1][x], m[y][x+1]);
9      } // Für die Länge: return m[0][0];
10 string res;
11 while(x < b.length() && y < a.length()) {
12     if(a[y] == b[x]) res += a[y++], x++;
13     else if(m[y][x+1] > m[y+1][x+1]) x++;
14     else y++;
15 }
16 return res;
17 }

```

## 5.8 Rolling Hash

```

1  ll q = 31; // Größer als Alphabetgröße. q=31,53,311
2  struct Hasher {
3      string s;
4      ll mod;
5      vector<ll> power, pref;
6      Hasher(const string& s, ll mod) : s(s), mod(mod) {
7          power.push_back(1);
8          for (int i = 1; i < (int)s.size(); i++)
9              power.push_back(power.back() * q % mod);
10         pref.push_back(0);
11         for (int i = 0; i < (int)s.size(); i++)
12             pref.push_back((pref.back() * q % mod + s[i]) % mod);
13     }
14
15     // Berechnet hash(s[l..r]). l,r inklusive.
16     ll hash(int l, int r) {
17         return (pref[r+1] - power[r-l+1] * pref[l] % mod + mod) % mod;
18     }
19 };

```

## 5.9 MANACHER's Algorithm, Longest Palindrome

```

1  char input[MAX_N];
2  char s[2 * MAX_N + 1];
3  int longest[2 * MAX_N + 1];
4
5  void setDots() {
6      s[0] = '.';
7      int j = 1;
8      for (int i = 0; i < (int)strlen(input); i++) {
9          s[j++] = input[i];
10         s[j++] = '.';
11     }
12     s[j] = '\0';
13 }
14
15 void manacher() {
16     int center = 0, last = 0, n = strlen(s);
17     memset(longest, 0, sizeof(longest));
18 }

```

```

19 for (int i = 1; i < n - 1; i++) {
20     int i2 = 2 * center - i;
21     longest[i] = (last > i) ? min(last - i, longest[i2]) : 0;
22     while (i + longest[i] + 1 < n && i - longest[i] - 1 >= 0 &&
23         s[i + longest[i] + 1] == s[i - longest[i] - 1]) longest[i]++;
24     if (i + longest[i] > last) {
25         center = i;
26         last = i + longest[i];
27     }
28 }
29 for (int i = 0; i < n; i++) longest[i] = 2 * longest[i] + 1;
30 }

```

## 6 Sonstiges

### 6.1 Zeileneingabe

```

1  // Zerlegt s anhand aller Zeichen in delim.
2  vector<string> split(string &s, string delim) {
3      vector<string> result; char *token;
4      token = strtok((char*)s.c_str(), (char*)delim.c_str());
5      while (token != NULL) {
6          result.push_back(string(token));
7          token = strtok(NULL, (char*)delim.c_str());
8      }
9      return result;
10 }

```

### 6.2 Bit Operations

```

1  // Bit an Position j auslesen.
2  (a & (1 << j)) != 0
3  // Bit an Position j setzen.
4  a |= (1 << j)
5  // Bit an Position j löschen.
6  a &= ~(1 << j)
7  // Bit an Position j umkehren.
8  a ^= (1 << j)
9  // Wert des niedrigsten gesetzten Bits.
10 (a & -a)
11 // Setzt alle Bits auf 1.
12 a = -1
13 // Setzt die ersten n Bits auf 1. Achtung: Overflows.
14 a = (1 << n) - 1
15 // Iteriert über alle Teilmengen einer Bitmaske (außer der leeren Menge).
16 for (int subset = bitmask; subset > 0; subset = (subset - 1) & bitmask)
17     // Anzahl der gesetzten Bits.
18     int __builtin_popcount(unsigned int x);
19     int __builtin_popcountll(unsigned long long x);
20     // Anzahl der führenden 0-Bits.
21     int __builtin_clz(unsigned int x);
22     int __builtin_clzll(unsigned long long x);

```

### 6.3 Rekursiver Abstieg und Abstrakter Syntaxbaum

```

1  struct Token { // In globalem Vektor, Zugriff über globale Variable.
2      int type; // Definiere Konstanten.
3      double value;

```

```

4 Token(int type) : type(type) {}
5 Token(int type, int value) : type(type), value(value) {}
6 };
7
8 struct Expression { // Die folgenden Klassen nur für den AST.
9     virtual ~Expression() {};
10    virtual double getValue() = 0;
11 };
12
13 struct Atom : public Expression {
14     double value;
15     Atom(int value) : value(value) {};
16     double getValue() { return value; }
17 };
18
19 struct BinaryExpression : public Expression {
20     Expression *lhs, *rhs;
21     BinaryExpression(Expression *lhs, Expression *rhs) : lhs(lhs), rhs(rhs) {}
22     ~BinaryExpression() { delete lhs; delete rhs; }
23 };
24
25 struct Addition : public BinaryExpression {
26     Addition(Expression *lhs, Expression *rhs) : BinaryExpression(lhs, rhs) {}
27     double getValue() { return lhs->getValue() + rhs->getValue(); }
28 };
29
30 Expression* parseF() {
31     Expression *lhs;
32     switch(tokens[token].type) {
33         case NUMBER: return new Atom(tokens[token++].value);
34         case LEFT_PAR:
35             token++;
36             lhs = parseA();
37             token++;
38             return lhs;
39         default:
40             return NULL;
41     }
42 }
43
44 Expression* parseA(Expression *lhs) {
45     Expression *plus, *minus;
46     if (token >= (int)tokens.size()) return lhs;
47     switch(tokens[token].type) {
48         case ADDITION:
49             token++;
50             plus = new Addition(lhs, parseS());
51             return parseA(plus);
52         case SUBTRACTION:
53             token++;
54             minus = new Subtraction(lhs, parseS());
55             return parseA(minus);
56         default:
57             return lhs;
58     }
59 }
60
61 Expression* parseA() {

```

```

60 Expression *lhs = parseS(); return parseA(lhs);
61 }

```

## 6.4 Sonstiges

```

1 // Alles-Header.
2 #include <bits/stdc++.h>
3 // Schnelle Ein-/Ausgabe mit cin/cout.
4 ios::sync_with_stdio(false);
5 cin.tie(NULL);
6 // Set mit eigener Sortierfunktion.
7 set<point2, decltype(comp)> set1(comp);
8 // PI
9 #define PI (2*acos(0))
10 // STL-Debugging, Compiler flags.
11 -D_GLIBCXX_DEBUG
12 // 128-Bit Integer/Float. Zum Einlesen/Ausgeben in long long casten.
13 __int128, __float128

```

## 6.5 Josephus-Problem

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

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

```

1 int rotateLeft(int n) { // Der letzte Überlebende, 1-basiert.
2     for (int i = 31; i >= 0; i--)
3         if (n & (1 << i)) {
4             n &= ~(1 << i);
5             break;
6         }
7     n <<= 1; n++; return n;
8 }

```

**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:  $F(1, k) = 0$ .

```

1 int josephus(int n, int k) { // Der letzte Überlebende, 0-basiert.
2     if (n == 1) return 0;
3     return (josephus(n-1, k) + k) % n;
4 }

```

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

## 6.6 Gemischtes

- **JOHNSONS Reweighting Algorithmus:** Füge neue Quelle  $s$  hinzu, mit Kanten mit Gewicht 0 zu allen Knoten. Nutze BELLMAN-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 Gewicht  $c$  ein. Füge Quelle  $s$  hinzu, mit

Kanten zu allen Knoten mit Gewicht 0. Nutze BELLMAN-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  $s \rightarrow A$  mit Gewicht  $w(A)$  und Kanten  $B \rightarrow t$  mit Gewicht  $w(B)$  hinzu. Füge Kanten mit Kapazität  $\infty$  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  $s$  erreichbar sind und die Knoten in  $B$ , die von  $t$  erreichbar 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 (KUH, Seite 8)
- **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:

$$[X/G] = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

- **POLYA Counting:** Sei  $\pi$  eine Permutation der Menge  $X$ . Die Elemente von  $X$  können mit einer von  $m$  Farben gefärbt werden. Die Anzahl der Färbungen, die Fixpunkte von  $\pi$  sind, ist  $m^{\#(\pi)}$ , wobei  $\#(\pi)$  die Anzahl der Zyklen von  $\pi$  ist. Die Anzahl der Färbungen von Objekten einer Menge  $X$  mit  $m$  Farben unter einer Symmetriegruppe  $G$  ist gegeben durch:

$$[X/G] = \frac{1}{|G|} \sum_{g \in G} m^{\#(g)}$$

- **Verteilung von Primzahlen:** Für alle  $n \in \mathbb{N}$  gilt: Es 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  $\{x_1, \dots, x_n\}$  mit  $x_1 \leq \dots \leq x_n$ . 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  $u_x \rightarrow v_y$  hinzu. Wenn Matching zu langsam ist, versuche Struktur des Posets auszunutzen und evtl. anders eine maximale Antikette zu finden.

- **Mo's Algorithm:** SQRT-Decomposition auf  $n$  Intervall Queries  $[l, r]$ . Gruppiere Queries in  $\sqrt{n}$  Blöcke nach linker Grenze  $l$ . Sortiere nach Block und bei gleichem Block nach rechter Grenze  $r$ . Beantworte Queries offline durch schrittweise Vergrößern/Verkleinern des aktuellen Intervalls. Laufzeit:  $O(n \cdot \sqrt{n})$ . (Anzahl der Blöcke als Konstante in Code schreiben.)
- **Centroids of a Tree:** Ein *Centroid* ist ein Knoten, der einen Baum in Komponenten der maximalen Größe  $\lfloor \frac{|V|}{2} \rfloor$  splittet. Es kann 2 Centroids geben! **Centroid Decomposition:** Wähle zufälligen Knoten und mache DFS. Verschiebe ausgewählten Knoten in Richtung des tiefsten Teilbaums, bis Centroid gefunden. Entferne Knoten, mache rekursiv in Teilbäumen weiter. Laufzeit:  $O(|V| \log(|V|))$ .
- **Kreuzprodukt**

$$a \times b = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

## 6.7 Tipps & Tricks

- Run Tim Error:
  - Array-Grenzen überprüfen. Indizierung bei 0 oder bei 1 beginnen?
  - Abbruchbedingung bei Rekursion?
  - Evtl. Memory Limit Exceeded?
  - $n$  und  $m$  verwechselt?
- Gleitkommazahlen:
  - NaN? Evtl. ungültige Werte für mathematische Funktionen, z.B.  $\text{acos}(1.0000000000000001)$ ?
  - 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.
    - \*  $n = 0, n = -1, n = 1, n = 2^{31} - 1, n = -2^{31} = 2147483648$
    - \*  $n$  gerade/ungerade
    - \* Graph ist leer/enthält nur einen Knoten.
    - \* Graph ist Multigraph (enthält Schleifen/Mehrfachkanten).
    - \* Sind Kanten gerichtet/ungerichtet?
    - \* Polygon ist konkav/selbstschneidend.
  - Bei DP/Rekursion: Stimmt Basisfall?
  - Unsicher bei benutzten STL-Funktionen?