

## 1 Datenstrukturen

### 1.1 Segmentbaum

SegTree baut den Baum auf  $O(n)$   
 query findet Summe über  $[l, r)$   $O(\log(n))$   
 update ändert einen Wert  $O(\log(n))$

```
1 struct SegTree {
2     using T = ll;
3     int n;
4     vector<T> tree;
5     static constexpr T E = 0; // Neutral element for combine
6     SegTree(vector<T>& a) : n(sz(a)), tree(2 * n) {
7         //SegTree(int size, T val = E) : n(size), tree(2 * n, val) {
8             copy(all(a), tree.begin() + n);
9             for (int i = n - 1; i > 0; i--) { // remove for range update
10                 tree[i] = comb(tree[2 * i], tree[2 * i + 1]);
11             }
12             T comb(T a, T b) {return a + b;} // modify this + neutral
13             void update(int i, T val) {
14                 tree[i += n] = val; // apply update code
15                 while (i /= 2) tree[i] = comb(tree[2 * i], tree[2 * i + 1]);
16             }
17             T query(int l, int r) {
18                 T resL = E, resR = E;
19                 for (l += n, r += n; l < r; l /= 2, r /= 2) {
20                     if (l&1) resL = comb(resL, tree[l++]);
21                     if (r&1) resR = comb(tree[--r], resR);
22                 }
23                 return comb(resL, resR);
24             }
25             // OR: range update + point query, needs commutative comb
26             void modify(int l, int r, T val) {
27                 for (l += n, r += n; l < r; l /= 2, r /= 2) {
28                     if (l&1) tree[l] = comb(tree[l], val), l++;
29                     if (r&1) --r, tree[r] = comb(tree[r], val);
30                 }
31                 T query(int i) {
32                     T res = E;
33                     for (i += n; i > 0; i /= 2) res = comb(res, tree[i]);
34                     return res;
35                 }
36             };
37         };
38     };
39 }
```

#### 1.1.1 Lazy Propagation

Assignment modifications, sum queries

lower\_bound erster Index in  $[l, r) \geq x$  (erfordert max-combine)  $O(\log(n))$

```
1 struct SegTree {
2     using T = ll; using U = ll;
3     int n;
4     static constexpr T E = 0; // Neutral element for combine
5     static constexpr U UF = inf; // Unused value by updates
6     vector<T> tree;
7     int h;
8     vector<U> lazy;
9     vector<int> k; // size of segments (optional)
10 }
```

```
10 SegTree(const vector<T>& a) : n(sz(a) + 1), tree(2 * n, E),
11 //SegTree(int size, T def = E) : n(size + 1), tree(2 * n, def),
12     h(ceil(log2(n))), lazy(n, UF), k(2 * n, 1) {
13     copy(all(a), tree.begin() + n);
14     for (int i = n - 1; i > 0; i--) {
15         k[i] = 2 * k[2 * i];
16         tree[i] = comb(tree[2 * i], tree[2 * i + 1]);
17     }
18     T comb(T a, T b) {return a + b;} // Modify this + E
19     void apply(int i, U val) { // And this + UF
20         tree[i] = val * k[i];
21         if (i < n) lazy[i] = val; // Don't forget this
22     }
23     void push_down(int i) {
24         if (lazy[i] != UF) {
25             apply(2 * i, lazy[i]);
26             apply(2 * i + 1, lazy[i]);
27             lazy[i] = UF;
28         }
29     }
30     void push(int i) {
31         for (int s = h; s > 0; s--) push_down(i >> s);
32     }
33     void build(int i) {
34         while (i /= 2) {
35             push_down(i);
36             tree[i] = comb(tree[2 * i], tree[2 * i + 1]);
37         }
38     }
39     void update(int l, int r, U val) {
40         l += n, r += n;
41         int l0 = l, r0 = r;
42         push(l0), push(r0 - 1);
43         for (; l < r; l /= 2, r /= 2) {
44             if (l&1) apply(l++, val);
45             if (r&1) apply(--r, val);
46         }
47         build(l0), build(r0 - 1);
48     }
49     T query(int l, int r) {
50         l += n, r += n;
51         push(l), push(r - 1);
52         T resL = E, resR = E;
53         for (; l < r; l /= 2, r /= 2) {
54             if (l&1) resL = comb(resL, tree[l++]);
55             if (r&1) resR = comb(tree[--r], resR);
56         }
57         return comb(resL, resR);
58     }
59     // Optional:
60     int lower_bound(int l, int r, T x) {
61         l += n, r += n;
62         push(l), push(r - 1);
63         int a[64] = {}, lp = 0, rp = 64;
64         for (; l < r; l /= 2, r /= 2) {
65             if (l&1) a[lp++] = l++;
66             if (r&1) a[rp--] = --r;
67         }
68         return lp - rp;
69     }
70 }
```

```
65 }
66 for (int i : a) if (i != 0 && tree[i] >= x) { // Modify this
67     while (i < n) {
68         push_down(i);
69         if (tree[2 * i] >= x) i = 2 * i; // And this
70         else i = 2 * i + 1;
71     }
72     return i - n;
73 }
74 return -1;
75 }
76 };
```

### 1.2 Wavelet Tree

WaveletTree baut den Baum auf  $O(n \log(\Sigma))$   
 kth sort  $[l, r][k]$   $O(\log(\Sigma))$   
 countSmaller Anzahl elemente in  $[l, r)$  kleiner als  $k$   $O(\log(\Sigma))$

```
1 struct WaveletTree {
2     using it = vector<ll>::iterator;
3     WaveletTree *ln = nullptr, *rn = nullptr;
4     vector<int> b = {0};
5     ll lo, hi;
6     WaveletTree(vector<ll> in) : WaveletTree(all(in)) {}
7     WaveletTree(it from, it to) : // call above one
8         lo(*min_element(from, to)), hi(*max_element(from, to) + 1) {
9         ll mid = (lo + hi) / 2;
10        auto f = [&](ll x) {return x < mid;};
11        for (it c = from; c != to; c++) {
12            b.push_back(b.back() + f(*c));
13        }
14        if (lo + 1 >= hi) return;
15        it pivot = stable_partition(from, to, f);
16        ln = new WaveletTree(from, pivot);
17        rn = new WaveletTree(pivot, to);
18    }
19    // kth element in sort[l, r) all 0-indexed
20    ll kth(int l, int r, int k) {
21        if (k < 0 || l + k >= r) return -1;
22        if (lo + 1 >= hi) return lo;
23        int inLeft = b[r] - b[l];
24        if (k < inLeft) return ln->kth(b[l], b[r], k);
25        else return rn->kth(l - b[l], r - b[r], k - inLeft);
26    }
27    // count elements in[l, r) smaller than k
28    int countSmaller(int l, int r, ll k) {
29        if (l >= r || k <= lo) return 0;
30        if (hi <= k) return r - l;
31        return ln->countSmaller(b[l], b[r], k) +
32            rn->countSmaller(l - b[l], r - b[r], k);
33    }
34    ~WaveletTree() {delete ln; delete rn;}
35 };
```

### 1.3 Fenwick Tree

init baut den Baum auf  $O(n \cdot \log(n))$   
 prefix\_sum summe von  $[0, i]$   $O(\log(n))$   
 update addiert ein Delta zu einem Element  $O(\log(n))$

```
1 vector<ll> tree;
2 void update(int i, ll val) {
3     for (i++; i < sz(tree); i += i & -i) tree[i] += val;
4 }
5 void init(int n) {
6     tree.assign(n + 1, 0);
7 }
8 ll prefix_sum(int i) {
9     ll sum = 0;
10    for (i++; i > 0; i -= i & -i) sum += tree[i];
11    return sum;
12 }
```

init baut den Baum auf  $O(n \cdot \log(n))$   
 prefix\_sum summe von  $[0, i]$   $O(\log(n))$   
 update addiert ein Delta zu allen Elementen  $[l, r]$ .  $l \leq r!$   $O(\log(n))$

```
1 vector<ll> add, mul;
2 void update(int l, int r, ll val) {
3     for (int tl = l + 1; tl < sz(add); tl += tl & -tl)
4         add[tl] += val, mul[tl] -= val * l;
5     for (int tr = r + 1; tr < sz(add); tr += tr & -tr)
6         add[tr] -= val, mul[tr] += val * r;
7 }
8 void init(vector<ll> &v) {
9     mul.assign(sz(v) + 1, 0);
10    add.assign(sz(v) + 1, 0);
11    for (int i = 0; i < sz(v); i++) update(i, i + 1, v[i]);
12 }
13 ll prefix_sum(int i) {
14     ll res = 0; i++;
15     for (int ti = i; ti > 0; ti -= ti & -ti)
16         res += add[ti] * i + mul[ti];
17     return res;
18 }
```

### 1.4 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++)
```

### 1.5 (Implicit) Treap (Cartesian Tree)

insert fügt wert *val* an stelle *i* ein (verschiebt alle Positionen  $\geq i$ )  $O(\log(n))$   
 remove löscht werte  $[i, i+count)$   $O(\log(n))$

```
1 mt19937 rng(0xc4bd5dad);
2 struct Treap {
3     struct Node {
4         ll val;
5         int prio, size = 1, l = -1, r = -1;
6         Node(ll x) : val(x), prio(rng()) {}
7     };
8     vector<Node> treap;
9     int root = -1;
10    int getSize(int v) {
11        return v < 0 ? 0 : treap[v].size;
12    }
13    void upd(int v) {
14        if (v < 0) return;
15        auto& V = treap[v];
16        V.size = 1 + getSize(V.l) + getSize(V.r);
17        // Update Node Code
18    }
19    void push(int v) {
20        if (v < 0) return;
21        //auto& V = treap[v];
22        //if (V.lazy) {
23            // Lazy Propagation Code
24            // if (V.l >= 0) treap[V.l].lazy = true;
25            // if (V.r >= 0) treap[V.r].lazy = true;
26            // V.lazy = false;
27            //}
28    }
29    pair<int, int> split(int v, int k) {
30        if (v < 0) return {-1, -1};
31        auto& V = treap[v];
32        push(v);
33        if (getSize(V.l) >= k) { // "V.val >= k" for lower_bound(k)
34            auto [left, right] = split(V.l, k);
35            V.l = right;
36            upd(v);
37            return {left, v};
38        } else {
39            // and only "k"
40            auto [left, right] = split(V.r, k - getSize(V.l) - 1);
41            V.r = left;
42            upd(v);
43            return {v, right};
44        }
45    }
46    int merge(int left, int right) {
47        if (left < 0) return right;
48        if (right < 0) return left;
49        if (treap[left].prio < treap[right].prio) {
50            push(left);
51            treap[left].r = merge(treap[left].r, right);
52            upd(left);
53            return left;
54        }
```

```
53     } else {
54         push(right);
55         treap[right].l = merge(left, treap[right].l);
56         upd(right);
57         return right;
58     }
59 void insert(int i, ll val) { // and i = val
60     auto [left, right] = split(root, i);
61     treap.emplace_back(val);
62     left = merge(left, sz(treap) - 1);
63     root = merge(left, right);
64 }
65 void remove(int i, int count = 1) {
66     auto [left, t_right] = split(root, i);
67     auto [middle, right] = split(t_right, count);
68     root = merge(left, right);
69 }
70 // for query use remove and read middle BEFORE remerging
71 };
```

### 1.6 Range Minimum Query

init baut Struktur auf  $O(n \cdot \log(n))$   
 queryIdempotent Index des Minimums in  $[l, r]$ .  $l < r!$   $O(1)$   
 • better-Funktion muss idempotent sein!

```
1 struct SparseTable {
2     vector<vector<int>> st;
3     ll *a;
4     int better(int lidx, int ridx) {
5         return a[lidx] <= a[ridx] ? lidx : ridx;
6     }
7     void init(vector<ll> &vec) {
8         int n = sz(*vec);
9         a = vec->data();
10        st.assign(__lg(n) + 1, vector<int>(n));
11        iota(all(st[0]), 0);
12        for (int j = 0; (2 <= j) <= n; j++) {
13            for (int i = 0; i + (2 <= j) <= n; i++) {
14                st[j + 1][i] = better(st[j][i], st[j][i + (1 <= j)]);
15            }
16        }
17        int queryIdempotent(int l, int r) {
18            if (r <= l) return -1;
19            int j = __lg(r - l); //31 - builtin_clz(r - l);
20            return better(st[j][l], st[j][r - (1 <= j)]);
21        }
22    };
```

### 1.7 STL-Bitset

```
1 bitset<10> bits(0b000010100);
2 bits._Find_first(); //2
3 bits._Find_next(2); //4
4 bits._Find_next(4); //10 bzw. N
5 bits[x] = 1; //not bits.set(x) or bits.reset(x)!
6 bits[x].flip(); //not bits.flip(x)!
7 bits.count(); //number of set bits
```

## 1.8 Link-Cut-Tree

LCT	baut Wald auf	$O(n)$
connected	prüft ob zwei Knoten im selben Baum liegen	$O(\log(n))$
link	fügt $\{x,y\}$ Kante ein	$O(\log(n))$
cut	entfernt $\{x,y\}$ Kante	$O(\log(n))$
lca	berechnet LCA von $x$ und $y$	$O(\log(n))$
query	berechnet query auf den Knoten des $xy$ -Pfades	$O(\log(n))$
modify	erhöht jeden wert auf dem $xy$ -Pfad	$O(\log(n))$

```

1 constexpr ll queryDefault = 0;
2 constexpr ll updateDefault = 0;
3 ll _modify(ll x, ll y) {
4     return x + y;
5 }
6 ll _query(ll x, ll y) {
7     return x + y;
8 }
9 ll _update(ll delta, int length) {
10     if (delta == updateDefault) return updateDefault;
11     //ll result = delta
12     //for (int i=1; i<length; i++) result = _query(result, delta);
13     return delta * length;
14 }
15 //generic:
16 ll joinValueDelta(ll value, ll delta) {
17     if (delta == updateDefault) return value;
18     return _modify(value, delta);
19 }
20 ll joinDeltas(ll delta1, ll delta2) {
21     if (delta1 == updateDefault) return delta2;
22     if (delta2 == updateDefault) return delta1;
23     return _modify(delta1, delta2);
24 }
25 struct LCT {
26     struct Node {
27         ll nodeValue, subTreeValue, delta;
28         bool revert;
29         int id, size;
30         Node *left, *right, *parent;
31
32         Node(int id = 0, int val = queryDefault) :
33             nodeValue(val), subTreeValue(val), delta(updateDefault),
34             revert(false), id(id), size(1),
35             left(nullptr), right(nullptr), parent(nullptr) {}
36
37         bool isRoot() {
38             return !parent || (parent->left != this &&
39                             parent->right != this);
40         }
41
42         void push() {
43             if (revert) {
44                 revert = false;
45                 swap(left, right);
46                 if (left) left->revert ^= 1;
47                 if (right) right->revert ^= 1;
48             }
49             nodeValue = joinValueDelta(nodeValue, delta);

```

```

47     subTreeValue = joinValueDelta(subTreeValue,
48                                   _update(delta, size));
49     if (left) left->delta = joinDeltas(left->delta, delta);
50     if (right) right->delta = joinDeltas(right->delta, delta);
51     delta = updateDefault;
52 }
53 ll getSubtreeValue() {
54     return joinValueDelta(subTreeValue, _update(delta, size));
55 }
56 void update() {
57     subTreeValue = joinValueDelta(nodeValue, delta);
58     size = 1;
59     if (left) {
60         subTreeValue = _query(subTreeValue,
61                               left->getSubtreeValue());
62         size += left->size;
63     }
64     if (right) {
65         subTreeValue = _query(subTreeValue,
66                               right->getSubtreeValue());
67         size += right->size;
68     }
69 };
70 vector<Node> nodes;
71 LCT(int n) : nodes(n) {
72     for (int i = 0; i < n; i++) nodes[i].id = i;
73 }
74 void connect(Node* ch, Node* p, int isLeftChild) {
75     if (ch) ch->parent = p;
76     if (isLeftChild >= 0) {
77         if (isLeftChild) p->left = ch;
78         else p->right = ch;
79     }
80 void rotate(Node* x) {
81     Node* p = x->parent;
82     Node* g = p->parent;
83     bool isRootP = p->isRoot();
84     bool leftChildX = (x == p->left);
85     connect(leftChildX ? x->right : x->left, p, leftChildX);
86     connect(p, x, !leftChildX);
87     connect(x, g, isRootP ? -1 : p == g->left);
88     p->update();
89 }
90 void splay(Node* x) {
91     while (!x->isRoot()) {
92         Node* p = x->parent;
93         Node* g = p->parent;
94         if (!p->isRoot()) g->push();
95         p->push();
96         x->push();
97         if (!p->isRoot()) rotate((x == p->left) ==
98                               (p == g->left) ? p : x);
99         rotate(x);
100     }
101     x->push();

```

```

102     x->update();
103 }
104 Node* expose(Node* x) {
105     Node* last = nullptr;
106     for (Node* y = x; y; y = y->parent) {
107         splay(y);
108         y->left = last;
109         last = y;
110     }
111     splay(x);
112     return last;
113 }
114 void makeRoot(Node* x) {
115     expose(x);
116     x->revert ^= 1;
117 }
118 bool connected(Node* x, Node* y) {
119     if (x == y) return true;
120     expose(x);
121     expose(y);
122     return x->parent;
123 }
124 void link(Node* x, Node* y) {
125     assert(!connected(x, y)); // not yet connected!
126     makeRoot(x);
127     x->parent = y;
128 }
129 void cut(Node* x, Node* y) {
130     makeRoot(x);
131     expose(y);
132     //must be a tree edge!
133     assert(!(y->right != x || x->left != nullptr));
134     y->right->parent = nullptr;
135     y->right = nullptr;
136 }
137 Node* lca(Node* x, Node* y) {
138     assert(connected(x, y));
139     expose(x);
140     return expose(y);
141 }
142 ll query(Node* from, Node* to) {
143     makeRoot(from);
144     expose(to);
145     if (to) return to->getSubtreeValue();
146     return queryDefault;
147 }
148 void modify(Node* from, Node* to, ll delta) {
149     makeRoot(from);
150     expose(to);
151     to->delta = joinDeltas(to->delta, delta);
152 }
153 };

```

## 1.9 Lichao

```

1 vector<ll> xs; // IMPORTANT: Initialize before constructing!
2 int findX(int i) {return lower_bound(all(xs), i) - begin(xs);}
3 struct Fun { // Default: Linear function. Change as needed.
4     ll m, c;
5     ll operator()(int x) {return m*xs[x] + c;}
6 };
7 // Default: Computes min. Change lines with comment for max.
8 struct Lichao {
9     static constexpr Fun id = {0, inf}; // {0, -inf}
10    int n, cap;
11    vector<Fun> seg;
12    Lichao() : n(sz(xs)), cap(2 << __lg(n)), seg(2 * cap, id) {}
13    void _insert(Fun f, int l, int r, int i) {
14        while (i < 2 * cap) {
15            int m = (l+r)/2;
16            if (m >= n) {r = m; i = 2*i; continue;}
17            Fun &g = seg[i];
18            if (f(m) < g(m)) swap(f, g); // >
19            if (f(l) < g(l)) r = m, i = 2*i; // >
20            else l = m, i = 2*i+1;
21        }
22        void insert(Fun f) {_insert(f, 0, cap, 1);}
23        void _segmentInsert(Fun f, int l, int r, int a, int b, int i) {
24            if (l <= a && b <= r) _insert(f, a, b, i);
25            else if (a < r && l < b) {
26                int m = (a+b)/2;
27                _segmentInsert(f, l, r, a, m, 2*i);
28                _segmentInsert(f, l, r, m, b, 2*i+1);
29            }
30            void segmentInsert(Fun f, ll l, ll r) {
31                _segmentInsert(f, findX(l), findX(r), 0, cap, 1);
32            }
33            ll _query(int x) {
34                ll ans = inf; // -inf
35                for (int i = x + cap; i > 0; i /= 2) {
36                    ans = min(ans, seg[i](x)); // max
37                }
38                return ans;
39            }
40            ll query(ll x) {return _query(findX(x));}
41 };

```

## 1.10 Policy Based Data Structures

Wichtig: Verwende p.swap(p2) anstatt swap(p, p2)!

```

1 #include <ext/pb_ds/priority_queue.hpp>
2 template<typename T>
3 using pQueue = __gnu_pbds::priority_queue<T>; //<T, greater<T>>
4 auto it = pq.push(5);
5 pq.modify(it, 6);
6 pq.join(pq2);
7 // push, join are O(1), pop, modify, erase O(log n) amortized

```

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 using namespace __gnu_pbds;
3 template<typename T>
4 using Tree = tree<T, null_type, less<T>, rb_tree_tag,
5     tree_order_statistics_node_update>;
6 // T.order_of_key(x): number of elements strictly less than x
7 // *T.find_by_order(k): k-th element
8 template<typename T>
9 struct chash {
10     static const
11         uint64_t C = ll(2e18 * acos(-1)) | 199; // random odd
12     size_t operator()(T o) const {
13         return __builtin_bswap64(hash<T>()(o) * C);
14     };
15 template<typename K, typename V>
16 using hashMap = gp_hash_table<K, V, chash<K>>;
17 template<typename T>
18 using hashSet = gp_hash_table<T, null_type, chash<T>>;

```

## 1.11 Lower/Upper Envelope (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 struct Line {
4     ll m, c;
5     ll operator()(ll x) {return m*x+c;}
6 };
7 vector<Line> ls;
8 ll ptr = 0;
9 bool bad(Line l1, Line l2, Line l3) {
10     return (l3.c-l1.c)*(l1.m-l2.m) < (l2.c-l1.c)*(l1.m-l3.m);
11 }
12 void add(ll m, ll c) { // Laufzeit O(1) amortisiert
13     while (sz(ls) > 1 && bad(ls.end()[-2], ls.end()[-1], {m, c})) {
14         ls.pop_back();
15     }
16     ls.push_back({m, c});
17     ptr = min(ptr, sz(ls) - 1);
18 }
19 ll query(ll x) { // Laufzeit: O(1) amortisiert
20     ptr = min(ptr, sz(ls) - 1);
21     while (ptr + 1 < sz(ls) && ls[ptr + 1](x) < ls[ptr](x)) ptr++;
22     return ls[ptr](x);
23 }

```

```

9 bool isect(iterator x, iterator y) {
10     if (y == end()) {x->p = INF; return false;}
11     if (x->m == y->m) x->p = x->b > y->b ? INF : -INF;
12     else x->p = div(y->b - x->b, x->m - y->m);
13     return x->p >= y->p;
14 }
15 void add(ll m, ll b) {
16     auto x = insert({m, b, 0});
17     while (isect(x, next(x))) erase(next(x));
18     if (x != begin()) {
19         x--;
20         if (isect(x, next(x))) {
21             erase(next(x));
22             isect(x, next(x));
23         }
24     }
25     while (x != begin() && prev(x)->p >= x->p) {
26         x--;
27         isect(x, erase(next(x)));
28     }
29     ll query(ll x) {
30         auto l = *lower_bound(x);
31         return l.m * x + l.b;
32     }
33 };

```

## 1.12 Union-Find

init	legt $n$ einzelne Unions an	$O(n)$
findSet	findet den Repräsentanten	$O(\log(n))$
unionSets	vereint 2 Mengen	$O(\log(n))$
$m \cdot \text{findSet} + n \cdot \text{unionSets}$	Folge von Befehlen	$O(n + m \cdot \alpha(n))$

```

1 // unions[i] >= 0 => unions[i] = parent
2 // unions[i] < 0 => unions[i] = -size
3 vector<int> unions;
4 void init(int n) { //Initialisieren
5     unions.assign(n, -1);
6 }
7 int findSet(int a) { // Pfadkompression
8     if (unions[a] < 0) return a;
9     return unions[a] = findSet(unions[a]);
10 }
11 void linkSets(int a, int b) { // Union by size.
12     if (unions[b] > unions[a]) swap(a, b);
13     unions[b] += unions[a];
14     unions[a] = b;
15 }
16 void unionSets(int a, int b) { // Diese Funktion aufrufen.
17     if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
18 }
19 int size(int a) {
20     return -unions[findSet(a)];
21 }

```

### 1.13 Persistent

get berechnet Wert zu Zeitpunkt  $t$   $O(\log(t))$   
 set ändert Wert zu Zeitpunkt  $t$   $O(\log(t))$   
 reset setzt die Datenstruktur auf Zeitpunkt  $t$   $O(1)$

```
1 template<typename T>
2 struct persistent {
3     int& time;
4     vector<pair<int, T>> data;
5     persistent(int& time, T value = {})
6         : time(time), data(1, {2*time, value}) {}
7     T get(int t) {
8         return prev
9             (upper_bound(all(data), pair{2*t+1, T{}}))->second;
10    }
11    int set(T value) {
12        time++;
13        data.push_back({2*time, value});
14        return time;
15    }
16 };

```

```
1 template<typename T>
2 struct persistentArray {
3     int time;
4     vector<persistent<T>> data;
5     vector<pair<int, int>> mods;
6     persistentArray(int n, T value = {})
7         : time(0), data(n, {time, value}) {}
8     T get(int p, int t) {return data[p].get(t);}
9     int set(int p, T value) {
10        mods.push_back({p, data[p].set(value)});
11        return mods.back().second;
12    }
13    void reset(int t) {
14        while (!mods.empty() && mods.back().second > t) {
15            data[mods.back().first].data.pop_back();
16            mods.pop_back();
17        }
18        time = t;
19    }
20 };

```

## 2 Graphen

### 2.1 Kruskal

berechnet den Minimalen Spannbaum  $O(|E| \cdot \log(|E|))$

```
1 sort(all(edges));
2 vector<Edge> mst;
3 ll cost = 0;
4 for (Edge& e : edges) {
5     if (findSet(e.from) != findSet(e.to)) {
6         unionSets(e.from, e.to);
7         mst.push_back(e);
8         cost += e.cost;
9     }
10 }

```

```
9 }}

```

### 2.2 Minimale Spannbäume

**Schnitteigenschaft** Für jeden Schnitt  $C$  im Graphen gilt: Gibt es eine Kante  $e$ , die echt leichter ist als alle anderen Schnittkanten, so gehört diese zu allen minimalen Spannbäumen. ( $\Rightarrow$  Die leichteste Kante in einem Schnitt kann in einem minimalen Spannbaum verwendet werden.)

**Kreiseigenschaft** Für jeden Kreis  $K$  im Graphen gilt: Die schwerste Kante auf dem Kreis ist nicht Teil des minimalen Spannbau.

### 2.3 Heavy-Light Decomposition

get\_intervals gibt Zerlegung des Pfades von  $u$  nach  $v$   $O(\log(|V|))$

**Wichtig:** Intervalle sind halboffen  
 Subbaum unter dem Knoten  $v$  ist das Intervall  $[in[v], out[v])$ .

```
1 vector<vector<int>> adj;
2 vector<int> sz, in, out, nxt, par;
3 int counter;
4 void dfs_sz(int v = 0, int from = -1) {
5     for (auto& u : adj[v]) if (u != from) {
6         dfs_sz(u, v);
7         sz[v] += sz[u];
8         if (adj[v][0] == from || sz[u] > sz[adj[v][0]]) {
9             swap(u, adj[v][0]); //changes adj!
10    }
11 }
12 void dfs_hld(int v = 0, int from = -1) {
13     par[v] = from;
14     in[v] = counter++;
15     for (int u : adj[v]) if (u != from) {
16         nxt[u] = (u == adj[v][0]) ? nxt[v] : u;
17         dfs_hld(u, v);
18     }
19     out[v] = counter;
20 }
21 void init(int root = 0) {
22     int n = sz(adj);
23     sz.assign(n, 1), nxt.assign(n, root), par.assign(n, -1);
24     in.resize(n), out.resize(n);
25     counter = 0;
26     dfs_sz(root);
27     dfs_hld(root);
28 }
29 template<typename F>
30 void for_intervals(int u, int v, F& f) {
31     for (; v = par[nxt[v]]; f(v)) {
32         if (in[v] < in[u]) swap(u, v);
33         f(max(in[u], in[nxt[v]]), in[v] + 1);
34         if (in[nxt[v]] <= in[u]) return;
35     }
36 }
37 int get_lca(int u, int v) {
38     for (; v = par[nxt[v]]; f(v)) {
39         if (in[v] < in[u]) swap(u, v);
40         if (in[nxt[v]] <= in[u]) return u;
41     }
42 }

```

### 2.4 Lowest Common Ancestor

init baut DFS-Baum über  $g$  auf  $O(|V| \cdot \log(|V|))$   
 getLCA findet LCA  $O(1)$   
 getDepth berechnet Distanz zur Wurzel im DFS-Baum  $O(1)$

```
1 struct LCA {
2     vector<ll> depth;
3     vector<int> visited, first;
4     int idx;
5     SparseTable st; //sparse table Seite 2
6     void init(vector<vector<int>>& adj, int root) {
7         depth.assign(2 * sz(adj), 0);
8         visited.assign(2 * sz(adj), -1);
9         first.assign(sz(adj), 2 * sz(adj));
10        idx = 0;
11        dfs(adj, root);
12        st.init(&depth);
13    }
14    void dfs(vector<vector<int>>& adj, int v, ll d=0) {
15        visited[idx] = v, depth[idx] = d;
16        first[v] = min(idx, first[v]), idx++;
17        for (int u : adj[v]) {
18            if (first[u] == 2 * sz(adj)) {
19                dfs(adj, u, d + 1);
20                visited[idx] = v, depth[idx] = d, idx++;
21            }
22        }
23        int getLCA(int u, int v) {
24            if (first[u] > first[v]) swap(u, v);
25            return visited[st.queryIdempotent(first[u], first[v] + 1)];
26        }
27        ll getDepth(int v) {return depth[first[v]];}
28    };

```

### 2.5 Centroids

find\_centroid findet alle Centroids des Baums (maximal 2)  $O(|V|)$

```
1 vector<int> s;
2 void dfs_sz(int v, int from = -1) {
3     s[v] = 1;
4     for (int u : adj[v]) if (u != from) {
5         dfs_sz(u, v);
6         s[v] += s[u];
7     }
8 }
9 pair<int, int> dfs_cent(int v, int from, int n) {
10     for (int u : adj[v]) if (u != from) {
11         if (2 * s[u] == n) return {v, u};
12         if (2 * s[u] > n) return dfs_cent(u, v, n);
13     }
14     return {v, -1};
15 }
16 pair<int, int> find_centroid(int root = 0) {
17     s.resize(sz(adj));
18     dfs_sz(root);
19     return dfs_cent(root, -1, s[root]);
20 }

```



## 2.6 Eulertouren

euler berechnet den Kreis  $O(|V|+|E|)$

```

1 vector<vector<int>> idx;
2 vector<int> to, validIdx, cycle;
3 vector<bool> used;
4
5 void addEdge(int u, int v) {
6     idx[u].push_back(sz(to));
7     to.push_back(v);
8     used.push_back(false);
9     idx[v].push_back(sz(to)); // für ungerichtet
10    to.push_back(u);
11    used.push_back(false);
12}
13
14 void euler(int v) { // init idx und validIdx
15     for (; validIdx[v] < sz(idx[v]); validIdx[v]++) {
16         if (!used[idx[v][validIdx[v]]]) {
17             int u = to[idx[v][validIdx[v]]];
18             used[idx[v][validIdx[v]]] = true;
19             used[idx[v][validIdx[v]] ^ 1] = true; // für ungerichtet
20             euler(u);
21         }
22     }
23     cycle.push_back(v); // Zyklus in umgekehrter Reihenfolge.
24 }
```

- Zyklus existiert, wenn jeder Knoten geraden Grad hat (ungerichtet), bei jedem Knoten Ein- und Ausgangsgrad übereinstimmen (gerichtet).
- Pfad existiert, wenn genau  $\{0,2\}$  Knoten ungeraden Grad haben (ungerichtet), bei allen Knoten Ein- und Ausgangsgrad übereinstimmen oder einer eine Ausgangskante mehr hat (Startknoten) und einer eine Eingangskante mehr hat (Endknoten).
- Je nach Aufgabenstellung überprüfen, wie ein unzusammenhängender Graph interpretiert werden sollen.
- Wenn eine bestimmte Sortierung verlangt wird oder Laufzeit vernachlässigbar ist, ist eine Implementierung mit einem `vector<set<int>>` adj leichter
- Wichtig: Algorithmus schlägt nicht fehl, falls kein Eulerzyklus existiert. Die Existenz muss separat geprüft werden.

## 2.7 Baum-Isomorphie

treeLabel berechnet kanonischen Namen für einen Baum  $O(|V|\log(|V|))$

```

1 vector<vector<int>> adj;
2 map<vector<int>, int> known; // dont reset!
3
4 int treeLabel(int v, int from = -1) {
5     vector<int> children;
6     for (int u : adj[v]) {
7         if (u == from) continue;
8         children.push_back(treeLabel(u, v));
9     }
10    sort(all(children));
11    if (known.find(children) == known.end()) {
12        known[children] = sz(known);
13    }
14    return known[children];
15 }
```

## 2.8 Kürzeste Wege

### 2.8.1 BELLMAN-FORD-Algorithmus

bellmanFord kürzeste Pfade oder negative Kreise finden  $O(|V|\cdot|E|)$

```

1 auto bellmanFord(int n, vector<edge>& edges, int start) {
2     vector<ll> dist(n, INF), prev(n, -1);
3     dist[start] = 0;
4
5     for (int i = 1; i < n; i++) {
6         for (edge& e : edges) {
7             if (dist[e.from] != INF &&
8                 dist[e.from] + e.cost < dist[e.to]) {
9                 dist[e.to] = dist[e.from] + e.cost;
10                prev[e.to] = e.from;
11            }
12        }
13    }
14    for (edge& e : edges) {
15        if (dist[e.from] != INF &&
16            dist[e.from] + e.cost < dist[e.to]) {
17            // Negativer Kreis gefunden.
18        }
19    }
20    return dist; //return prev?
21 }
```

### 2.8.2 Algorithmus von DIJKSTRA

dijkstra kürzeste Pfade in Graphen ohne negative Kanten  $O(|E|\log(|V|))$

```

1 using path = pair<ll, int>; //dist, destination
2
3 auto dijkstra(const vector<vector<path>>& adj, int start) {
4     priority_queue<path, vector<path>, greater<path>> pq;
5     vector<ll> dist(sz(adj), INF);
6     vector<int> prev(sz(adj), -1);
7     dist[start] = 0; pq.emplace(0, start);
8
9     while (!pq.empty()) {
10        auto [dv, v] = pq.top(); pq.pop();
11        if (dv > dist[v]) continue; // WICHTIG!
12
13        for (auto [du, u] : adj[v]) {
14            ll newDist = dv + du;
15            if (newDist < dist[u]) {
16                dist[u] = newDist;
17                prev[u] = v;
18                pq.emplace(dist[u], u);
19            }
20        }
21    }
22    return dist; //return prev;
23 }
```

### 2.8.3 FLOYD-WARSHALL-Algorithmus

floydWarshall kürzeste Pfade oder negative Kreise finden  $O(|V|^3)$

- $\text{dist}[i][i] = 0$ ,  $\text{dist}[i][j] = \text{edge}[j, i].\text{weight}$  oder  $\text{INF}$
- $i$  liegt auf einem negativen Kreis  $\Leftrightarrow \text{dist}[i][i] < 0$ .

```

1 vector<vector<ll>> dist; // Entfernung zwischen je zwei Punkten
2 vector<vector<int>> next;
3
4 void floydWarshall() {
5     next.assign(sz(dist), vector<int>(sz(dist), -1));
6     for (int i = 0; i < sz(dist); i++) {
7         for (int j = 0; j < sz(dist); j++) {
8             if (dist[i][j] < INF) {
9                 next[i][j] = j;
10            }
11        }
12    }
13 }
```

```

9     }}}
10
11     for (int k = 0; k < sz(dist); k++) {
12         for (int i = 0; i < sz(dist); i++) {
13             for (int j = 0; j < sz(dist); j++) {
14                 // only needed if dist can be negative
15                 if (dist[i][k] == INF || dist[k][j] == INF) continue;
16                 if (dist[i][j] > dist[i][k] + dist[k][j]) {
17                     dist[i][j] = dist[i][k] + dist[k][j];
18                     next[i][j] = next[i][k];
19                 }
20             }
21         }
22     }
23
24     vector<int> getPath(int u, int v) {
25         if (next[u][v] < 0) return {};
26         vector<int> path = {u};
27         while (u != v) path.push_back(u = next[u][v]);
28         return path; //Pfad u -> v
29     }
30 }
```

### 2.8.4 Matrix-Algorithmus

Sei  $d_{ij}$  die Distanzmatrix von  $G$ , dann gibt  $d_{ij}^k$  die kürzeste Distanz von  $i$  nach  $j$  mit maximal  $k$  Kanten an mit der Verknüpfung:  $c_{ij} = a_{ij} \otimes b_{ij} = \min\{a_{ik} \cdot b_{kj}\}$ . Sei  $a_{ij}$  die Adjazenzmatrix von  $G$  (mit  $a_{ii} = 1$ ), dann gibt  $a_{ij}^k$  die Anzahl der Wege von  $i$  nach  $j$  mit Länge genau (maximal)  $k$  an mit der Verknüpfung:  $c_{ij} = a_{ij} \otimes b_{ij} = \sum a_{ik} \cdot b_{kj}$ .

### 2.9 Dynamic Connectivity

Constructor erzeugt Baum ( $n$  Knoten,  $m$  Updates)  $O(n+m)$   
 addEdge fügt Kante ein, id=delete Zeitpunkt  $O(\log(n))$   
 eraseEdge entfernt Kante id  $O(\log(n))$

```

1 struct connect {
2     int n;
3     vector<pair<int, int>> edges;
4     LCT lct; // min LCT Seite 3, no updates required
5
6     connect(int n, int m) : n(n), edges(m), lct(n+m) {}
7
8     bool connected(int u, int v) {
9         return lct.connected(&lct.nodes[u], &lct.nodes[v]);
10    }
11
12    void addEdge(int u, int v, int id) {
13        lct.nodes[id + n] = LCT::Node(id + n, id);
14        edges[id] = {u, v};
15        if (connected(u, v)) {
16            int old = lct.query(&lct.nodes[u], &lct.nodes[v]);
17            if (old < id) eraseEdge(old);
18        }
19
20        if (!connected(u, v)) {
21            lct.link(&lct.nodes[u], &lct.nodes[id + n]);
22            lct.link(&lct.nodes[v], &lct.nodes[id + n]);
23        }
24    }
25
26    void eraseEdge(int id) {
27        if (connected(edges[id].first, edges[id].second) &&
28            lct.query(&lct.nodes[edges[id].first],
29                    &lct.nodes[edges[id].second]) == id) {
30            lct.cut(&lct.nodes[edges[id].first], &lct.nodes[id + n]);
31            lct.cut(&lct.nodes[edges[id].second], &lct.nodes[id + n]);
32        }
33    }
34 }
```

## 2.10 Erdős-Gallai

Sei  $d_1 \geq \dots \geq d_n$ . Es existiert genau dann ein Graph  $G$  mit Degree sequence

$d$  falls  $\sum_{i=1}^n d_i$  gerade ist und für  $1 \leq k \leq n$ :  $\sum_{i=1}^k d_i \leq k \cdot (k-1) + \sum_{i=k+1}^n \min(d_i, k)$

haveHakimi findet Graph  $O((|V|+|E|) \cdot \log(|V|))$

```
1 vector<vector<int>> haveHakimi(const vector<int>& deg) {
2     priority_queue<pair<int, int>> pq;
3     for (int i = 0; i < sz(deg); i++) {
4         if (deg[i] > 0) pq.push({deg[i], i});
5     }
6     vector<vector<int>> adj(sz(deg));
7     while (!pq.empty()) {
8         auto [degV, v] = pq.top(); pq.pop();
9         if (sz(pq) < degV) return {}; //impossible
10        vector<pair<int, int>> todo(degV);
11        for (auto& e : todo) e = pq.top(), pq.pop();
12        for (auto [degU, u] : todo) {
13            adj[v].push_back(u);
14            adj[u].push_back(v);
15            if (degU > 1) pq.push({degU - 1, u});
16        }
17        return adj;
18    }
```

## 2.11 Strongly Connected Components (TARJAN)

scc berechnet starke Zusammenhangskomponenten  $O(|V|+|E|)$

```
1 vector<vector<int>> adj, sccs;
2 int counter;
3 vector<bool> inStack;
4 vector<int> low, idx, s; //idx enthält Index der SCC pro Knoten.
5 void visit(int v) {
6     int old = low[v] = counter++;
7     s.push_back(v); inStack[v] = true;
8     for (auto u : adj[v]) {
9         if (low[u] < 0) visit(u);
10        if (inStack[u]) low[v] = min(low[v], low[u]);
11    }
12    if (old == low[v]) {
13        sccs.push_back({});
14        for (int u = -1; u != v; ) {
15            u = s.back(); s.pop_back(); inStack[u] = false;
16            idx[u] = sz(sccs) - 1;
17            sccs.back().push_back(u);
18        }
19    }
20    void scc() {
21        inStack.assign(sz(adj), false);
22        low.assign(sz(adj), -1);
23        idx.assign(sz(adj), -1);
24        sccs.clear();
25        counter = 0;
26        for (int i = 0; i < sz(adj); i++) {
27            if (low[i] < 0) visit(i);
28        }
```

## 2.12 DFS

Kantentyp $(v, w)$	$\text{dfs}[v] < \text{dfs}[w]$	$\text{fin}[v] > \text{fin}[w]$	seen[w]
in-tree	true	true	false
forward	true	true	true
backward	false	false	true
cross	false	true	true

## 2.13 Artikulationspunkte, Brücken und BCC

find berechnet Artikulationspunkte, Brücken und BCC  $O(|V|+|E|)$

Wichtig: isolierte Knoten und Brücken sind keine BCC.

```
1 vector<vector<Edge>> adj;
2 vector<int> num;
3 int counter, rootCount, root;
4 vector<bool> isArt;
5 vector<Edge> bridges, st;
6 vector<vector<Edge>> bcc;
7 int dfs(int v, int from = -1) {
8     int me = num[v] = ++counter, top = me;
9     for (Edge& e : adj[v]) {
10        if (e.id == from) continue;
11        if (num[e.to]) {
12            top = min(top, num[e.to]);
13            if (num[e.to] < me) st.push_back(e);
14        } else {
15            if (v == root) rootCount++;
16            int si = sz(st);
17            int up = dfs(e.to, e.id);
18            top = min(top, up);
19            if (up >= me) isArt[v] = true;
20            if (up > me) bridges.push_back(e);
21            if (up <= me) st.push_back(e);
22            if (up == me) {
23                bcc.emplace_back(si + all(st));
24                st.resize(si);
25            }
26        }
27        return top;
28    }
29 void find() {
30     counter = 0;
31     num.assign(sz(adj), 0);
32     isArt.assign(sz(adj), false);
33     bridges.clear();
34     st.clear();
35     bcc.clear();
36     for (int v = 0; v < sz(adj); v++) {
37         if (!num[v]) {
38             root = v;
39             rootCount = 0;
40             dfs(v);
41             isArt[v] = rootCount > 1;
42         }
43     }
```

## 2.14 2-SAT

```
1 struct sat2 {
2     int n; // + scc variablen
3     vector<int> sol;
4     sat2(int vars) : n(vars*2), adj(n) {}
5     static int var(int i) {return i << 1;} // use this!
6     void addImpl(int a, int b) {
7         adj[a].push_back(b);
8         adj[1^b].push_back(1^a);
9     }
10    void addEquiv(int a, int b) {addImpl(a, b); addImpl(b, a);}
11    void addOr(int a, int b) {addImpl(1^a, b);}
12    void addXor(int a, int b) {addOr(a, b); addOr(1^a, 1^b);}
13    void addTrue(int a) {addImpl(1^a, a);}
14    void addFalse(int a) {addTrue(1^a);}
15    void addAnd(int a, int b) {addTrue(a); addTrue(b);}
16    void addNand(int a, int b) {addOr(1^a, 1^b);}
17    bool solve() {
18        scc(); //scc code von oben
19        sol.assign(n, -1);
20        for (int i = 0; i < n; i += 2) {
21            if (idx[i] == idx[i + 1]) return false;
22            sol[i] = idx[i] < idx[i + 1];
23            sol[i + 1] = !sol[i];
24        }
25        return true;
26    }
27 };
```

## 2.15 Maximal Cliques

bronKerbosch berechnet alle maximalen Cliques  $O(3^{\frac{n}{2}})$

addEdge fügt ungerichtete Kante ein  $O(1)$

```
1 using bits = bitset<64>;
2 vector<bits> adj, cliques;
3 void addEdge(int a, int b) {
4     if (a != b) adj[a][b] = adj[b][a] = 1;
5 }
6 void bronKerboschRec(bits R, bits P, bits X) {
7     if (P.none() && X.none()) {
8         cliques.push_back(R);
9     } else {
10        int q = min(P._Find_first(), X._Find_first());
11        bits cands = P & ~adj[q];
12        for (int i = 0; i < sz(adj); i++) if (cands[i]) {
13            R[i] = 1;
14            bronKerboschRec(R, P & adj[i], X & adj[i]);
15            R[i] = P[i] = 0;
16            X[i] = 1;
17        }
18    }
19 void bronKerbosch() {
20     cliques.clear();
21     bronKerboschRec({}, {(1ull << sz(adj)) - 1}, {});
22 }
```

## 2.16 Cycle Counting

findBase berechnet Basis  $O(|V| \cdot |E|)$

count zählt Zykel  $O(2^{|base|})$

- jeder Zyklus ist das xor von einträgen in base.

```
1 constexpr int maxEdges = 128;
2 using cycle = bitset<maxEdges>;
3 struct cycles {
4     vector<vector<pair<int, int>>> adj;
5     vector<bool> seen;
6     vector<cycle> paths, base;
7     vector<pair<int, int>> edges;
8     cycles(int n) : adj(n), seen(n), paths(n) {}
9     void addEdge(int u, int v) {
10         adj[u].push_back({v, sz(edges)});
11         adj[v].push_back({u, sz(edges)});
12         edges.push_back({u, v});
13     }
14     void addBase(cycle cur) {
15         for (cycle o : base) {
16             o ^= cur;
17             if (o._Find_first() > cur._Find_first()) cur = o;
18         }
19         if (cur.any()) base.push_back(cur);
20     }
21     void findBase(int v, int from = -1, cycle cur = {}) {
22         if (from < 0 && seen[v]) return;
23         if (seen[v]) {
24             addBase(cur ^ paths[v]);
25         } else {
26             seen[v] = true;
27             paths[v] = cur;
28             for (auto [u, id] : adj[v]) {
29                 if (u == from) continue;
30                 cur[id].flip();
31                 findBase(u, v, cur);
32                 cur[id].flip();
33             }
34         }
35         bool isCycle(cycle cur) { // cycle must be constructed from base
36             if (cur.none()) return false;
37             init(sz(adj)); // union find Seite 4
38             for (int i = 0; i < sz(edges); i++) {
39                 if (cur[i]) {
40                     cur[i] = false;
41                     if (findSet(edges[i].first) ==
42                         findSet(edges[i].second)) break;
43                     unionSets(edges[i].first, edges[i].second);
44                 }
45             }
46             return cur.none();
47         }
48         int count() {
49             for (int i = 0; i < sz(adj); i++) findBase(i);
50             assert(sz(base) < 30);
51             int res = 0;
52             for (int i = 1; i < (1 << sz(base)); i++) {
53                 cycle cur;
```

```
52         for (int j = 0; j < sz(base); j++)
53             if (((i >> j) & 1) != 0) cur ^= base[j];
54             if (isCycle(cur)) res++;
55         }
56         return res;
57     }
58 };
```

## 2.17 Wert des maximalen Matchings

Fehlerwahrscheinlichkeit:  $(\frac{m}{MOD})^I$

```
1 constexpr int MOD=1'000'000'007, I=10;
2 vector<vector<ll>> adj, mat;
3 int max_matching() {
4     int ans = 0;
5     mat.assign(sz(adj), {});
6     for (int _ = 0; _ < I; _++) {
7         for (int v = 0; v < sz(adj); v++) {
8             mat[v].assign(sz(adj), 0);
9             for (int u : adj[v]) {
10                 if (u < v) {
11                     mat[v][u] = rand() % (MOD - 1) + 1;
12                     mat[u][v] = MOD - mat[v][u];
13                 }
14             }
15             gauss(sz(adj), MOD); // LGS Seite 17
16             int rank = 0;
17             for (auto& row : mat) {
18                 if (*max_element(all(row)) != 0) rank++;
19             }
20             ans = max(ans, rank / 2);
21         }
22     }
23     return ans;
24 }
```

## 2.18 Allgemeines maximales Matching

match berechnet allgemeines Matching  $O(|E| \cdot |V| \cdot \log(|V|))$

```
1 struct GM {
2     vector<vector<int>> adj;
3     // pairs ist der gematchte Knoten oder n
4     vector<int> pairs, first, que;
5     vector<pair<int, int>> label;
6     int head, tail;
7     GM(int n) : adj(n), pairs(n + 1, n), first(n + 1, n),
8                 que(n), label(n + 1, {-1, -1}) {}
9     void rematch(int u, int v) {
10         int t = pairs[u]; pairs[u] = v;
11         if (pairs[t] != u) return;
12         if (label[u].second == -1) {
13             pairs[t] = label[u].first;
14             rematch(pairs[t], t);
15         } else {
16             auto [x, y] = label[u];
17             rematch(x, y);
18             rematch(y, x);
19         }
20     }
21     int findFirst(int v) {
```

```
21         return label[first[v]].first < 0 ? first[v]
22             : first[v] = findFirst(first[v]);
23     }
24     void relabel(int x, int y) {
25         int r = findFirst(x);
26         int s = findFirst(y);
27         if (r == s) return;
28         auto h = label[r] = label[s] = {-x, y};
29         int join;
30         while (true) {
31             if (s != sz(adj)) swap(r, s);
32             r = findFirst(label[pairs[r]].first);
33             if (label[r] == h) {
34                 join = r;
35                 break;
36             } else {
37                 label[r] = h;
38             }
39         }
40         for (int v : {first[x], first[y]}) {
41             for (; v != join; v = first[label[pairs[v]].first]) {
42                 label[v] = {x, y};
43                 first[v] = join;
44                 que[tail++] = v;
45             }
46         }
47         bool augment(int v) {
48             label[v] = {sz(adj), -1};
49             first[v] = sz(adj);
50             head = tail = 0;
51             for (que[tail++] = v; head < tail;) {
52                 int x = que[head++];
53                 for (int y : adj[x]) {
54                     if (pairs[y] == sz(adj) && y != v) {
55                         pairs[y] = x;
56                         rematch(x, y);
57                         return true;
58                     } else if (label[y].first >= 0) {
59                         relabel(x, y);
60                     } else if (label[pairs[y]].first == -1) {
61                         label[pairs[y]].first = x;
62                         first[pairs[y]] = y;
63                         que[tail++] = pairs[y];
64                     }
65                 }
66             }
67             return false;
68         }
69         int match() {
70             int matching = head = tail = 0;
71             for (int v = 0; v < sz(adj); v++) {
72                 if (pairs[v] < sz(adj) || !augment(v)) continue;
73                 matching++;
74                 for (int i = 0; i < tail; i++) {
75                     label[que[i]] = label[pairs[que[i]]] = {-1, -1};
76                     label[sz(adj)] = {-1, -1};
77                 }
78             }
79             return matching;
80         }
81     };
82 }
```



## 2.19 Rerooting Template

```

1 // Usual Tree DP can be broken down in 4 steps:
2 // - Initialize dp[v] = identity
3 // - Iterate over all children w and take a value for w
4 //   by looking at dp[w] and possibly the edge label of v -> w
5 // - combine the values of those children
6 // - usually this operation should be commutative and associative
7 // - finalize the dp[v] after iterating over all children
8 struct Reroot {
9     using T = ll;
10    // identity element
11    T E() {}
12    // x: dp value of child
13    // e: index of edge going to child
14    T takeChild(T x, int e) {}
15    T comb(T x, T y) {}
16    // called after combining all dp values of children
17    T fin(T x, int v) {}
18    vector<vector<pair<int, int>>> g;
19    vector<int> ord, pae;
20    vector<T> dp;
21    T dfs(int v) {
22        ord.push_back(v);
23        for (auto [w, e] : g[v]) {
24            g[w].erase(find(all(g[w]), pair(v, e^1)));
25            pae[w] = e^1;
26            dp[w] = comb(dp[v], takeChild(dfs(w), e));
27        }
28        return dp[v] = fin(dp[v], v);
29    }
30    vector<T> solve(int n, vector<pair<int, int>> edges) {
31        g.resize(n);
32        for (int i = 0; i < n-1; i++) {
33            g[edges[i].first].emplace_back(edges[i].second, 2*i);
34            g[edges[i].second].emplace_back(edges[i].first, 2*i+1);
35        }
36        pae.assign(n, -1);
37        dp.assign(n, E());
38        dfs(0);
39        vector<T> updp(n, E()), res(n, E());
40        for (int v : ord) {
41            vector<T> pref(sz(g[v])+1), suff(sz(g[v])+1);
42            if (v != 0) pref[0] = takeChild(updp[v], pae[v]);
43            for (int i = 0; i < sz(g[v]); i++) {
44                auto [u, w] = g[v][i];
45                pref[i+1] = suff[i] = takeChild(dp[u], w);
46                pref[i+1] = comb(pref[i], pref[i+1]);
47            }
48            for (int i = sz(g[v])-1; i >= 0; i--) {
49                suff[i] = comb(suff[i], suff[i+1]);
50            }
51            for (int i = 0; i < sz(g[v]); i++) {
52                updp[g[v][i].first] = fin(comb(pref[i], suff[i+1]), v);
53            }
54            res[v] = fin(pref.back(), v);

```

```

55     }
56     return res;
57 }
58 };

```

## 2.20 Virtual Trees

```

1 // needs dfs in- and out- time and lca function
2 vector<int> in, out;
3 void virtualTree(vector<int> ind) { // indices of used nodes
4     sort(all(ind), [&](int x, int y) {return in[x] < in[y];});
5     for (int i = 0, n = sz(ind); i < n - 1; i++) {
6         ind.push_back(lca(ind[i], ind[i + 1]));
7     }
8     sort(all(ind), [&](int x, int y) {return in[x] < in[y];});
9     ind.erase(unique(all(ind), ind.end()));
10    int n = ind.size();
11    vector<vector<int>> tree(n);
12    vector<int> st = {0};
13    for (int i = 1; i < n; i++) {
14        while (in[ind[i]] >= out[ind[st.back()]]) st.pop_back();
15        tree[st.back()].push_back(i);
16        st.push_back(i);
17    }
18    // virtual directed tree with n nodes, original indices in ind
19    // weights can be calculated, e.g. with binary lifting
20 }

```

## 2.21 Maximum Cardinality Bipartite Matching

kuhn berechnet Matching  $O(|V| \cdot \min(ans^2, |E|))$   
 • die ersten  $[0..l)$  Knoten in adj sind die linke Seite des Graphen

```

1 vector<vector<int>> adj;
2 vector<int> pairs; // Der gematchte Knoten oder -1.
3 vector<bool> visited;
4 bool dfs(int v) {
5     if (visited[v]) return false;
6     visited[v] = true;
7     for (int u : adj[v]) if (pairs[u] < 0 || dfs(pairs[u])) {
8         pairs[u] = v; pairs[v] = u; return true;
9     }
10    return false;
11 }
12 int kuhn(int l) { // l = #Knoten links.
13     pairs.assign(sz(adj), -1);
14     int ans = 0;
15     // Greedy Matching. Optionale Beschleunigung.
16     for (int v = 0; v < l; v++) for (int u : adj[v])
17         if (pairs[u] < 0) {pairs[u] = v; pairs[v] = u; ans++; break;}
18     for (int v = 0; v < l; v++) if (pairs[v] < 0) {
19         visited.assign(l, false);
20         ans += dfs(v);
21     }
22     return ans; // Größe des Matchings.
23 }

```

hopcroft\_karp berechnet Matching  $O(\sqrt{|V|} \cdot |E|)$

```

1 vector<vector<int>> adj;
2 // pairs ist der gematchte Knoten oder -1
3 vector<int> pairs, dist, ptr;
4 bool bfs(int l) {
5     queue<int> q;
6     for(int v = 0; v < l; v++) {
7         if (pairs[v] < 0) {dist[v] = 0; q.push(v);}
8         else dist[v] = -1;
9     }
10    bool exist = false;
11    while(!q.empty()) {
12        int v = q.front(); q.pop();
13        for (int u : adj[v]) {
14            if (pairs[u] < 0) {exist = true; continue;}
15            if (dist[pairs[u]] < 0) {
16                dist[pairs[u]] = dist[v] + 1;
17                q.push(pairs[u]);
18            }
19        }
20    }
21    return exist;
22 }
23 bool dfs(int v) {
24     for (; ptr[v] < sz(adj[v]); ptr[v]++) {
25         int u = adj[v][ptr[v]];
26         if (pairs[u] < 0 || (dist[pairs[u]] > dist[v] && dfs(pairs[u]))) {
27             pairs[u] = v; pairs[v] = u;
28             return true;
29         }
30     }
31     return false;
32 }
33 int hopcroft_karp(int l) { // l = #Knoten links
34     int ans = 0;
35     pairs.assign(sz(adj), -1);
36     dist.resize(l);
37     // Greedy Matching, optionale Beschleunigung.
38     for (int v = 0; v < l; v++) for (int u : adj[v])
39         if (pairs[u] < 0) {pairs[u] = v; pairs[v] = u; ans++; break;}
40     while(bfs(l)) {
41         ptr.assign(l, 0);
42         for(int v = 0; v < l; v++) {
43             if (pairs[v] < 0) ans += dfs(v);
44         }
45     }
46     return ans;
47 }

```

## 2.22 Global Mincut

stoer\_wagner berechnet globalen Mincut  $O(|V||E| + |V|^2 \cdot \log(|E|))$

merge(a,b) merged Knoten b in Knoten a  $O(|E|)$

Tipp: Cut Rekonstruktion mit unionFind für Partitionierung oder vector<bool> für edge id's im cut.

```

1 struct Edge {
2     int from, to;
3     ll cap;
4 };

```

```

5 vector<vector<Edge>> adj, tmp;
6 vector<bool> erased;
7
8 void merge(int u, int v) {
9     tmp[u].insert(tmp[u].end(), all(tmp[v]));
10    tmp[v].clear();
11    erased[v] = true;
12    for (auto& vec : tmp) {
13        for (Edge& e : vec) {
14            if (e.from == v) e.from = u;
15            if (e.to == v) e.to = u;
16        }
17    }
18    ll stoer_wagner() {
19        ll res = INF;
20        tmp = adj;
21        erased.assign(sz(tmp), false);
22        for (int i = 1; i < sz(tmp); i++) {
23            int s = 0;
24            while (erased[s]) s++;
25            priority_queue<pair<ll, int>> pq;
26            pq.push({0, s});
27            vector<ll> con(sz(tmp));
28            ll cur = 0;
29            vector<pair<ll, int>> state;
30            while (!pq.empty()) {
31                int c = pq.top().second;
32                pq.pop();
33                if (con[c] < 0) continue; //already seen
34                con[c] = -1;
35                for (auto e : tmp[c]) {
36                    if (con[e.to] >= 0) { //add edge to cut
37                        con[e.to] += e.cap;
38                        pq.push({con[e.to], e.to});
39                        cur += e.cap;
40                    } else if (e.to != c) { //remove edge from cut
41                        cur -= e.cap;
42                    }
43                }
44                state.push_back({cur, c});
45            }
46            int t = state.back().second;
47            state.pop_back();
48            if (state.empty()) return 0; //graph is not connected?!
49            merge(state.back().second, t);
50            res = min(res, state.back().first);
51        }
52        return res;
53    }
54 }

```

## 2.23 Max-Flow

## 2.24 Min-Cost-Max-Flow

mincostflow berechnet Fluss  $O(|V|^2 \cdot |E|^2)$

```

1 constexpr ll INF = 1LL << 60; // Größer als der maximale Fluss.
2 struct MinCostFlow {
3     struct edge {
4         int to;
5         ll f, cost;
6     };

```

```

7     vector<edge> edges;
8     vector<vector<int>> adj;
9     vector<int> pref, con;
10    vector<ll> dist;
11    const int s, t;
12    ll maxflow, mincost;
13
14    MinCostFlow(int n, int source, int target) :
15        adj(n), s(source), t(target) {}
16
17    void addEdge(int u, int v, ll c, ll cost) {
18        adj[u].push_back(sz(edges));
19        edges.push_back({v, c, cost});
20        adj[v].push_back(sz(edges));
21        edges.push_back({u, 0, -cost});
22    }
23
24    bool SPFA() {
25        pref.assign(sz(adj), -1);
26        dist.assign(sz(adj), INF);
27        vector<bool> inqueue(sz(adj));
28        queue<int> queue;
29        dist[s] = 0;
30        queue.push(s);
31        pref[s] = s;
32        inqueue[s] = true;
33        while (!queue.empty()) {
34            int cur = queue.front(); queue.pop();
35            inqueue[cur] = false;
36            for (int id : adj[cur]) {
37                int to = edges[id].to;
38                if (edges[id].f > 0 &&
39                    dist[to] > dist[cur] + edges[id].cost) {
40                    dist[to] = dist[cur] + edges[id].cost;
41                    pref[to] = cur;
42                    con[to] = id;
43                    if (!inqueue[to]) {
44                        inqueue[to] = true;
45                        queue.push(to);
46                    }
47                }
48            }
49        }
50        return pref[t] != -1;
51    }
52
53    void extend() {
54        ll w = INF;
55        for (int u = t; pref[u] != u; u = pref[u])
56            w = min(w, edges[con[u]].f);
57        maxflow += w;
58        mincost += dist[t] * w;
59        for (int u = t; pref[u] != u; u = pref[u]) {
60            edges[con[u]].f -= w;
61            edges[con[u] ^ 1].f += w;
62        }
63    }
64
65    void mincostflow() {
66        con.assign(sz(adj), 0);
67        maxflow = mincost = 0;
68        while (SPFA()) extend();
69    }
70 }

```

## 2.24.1 Dinic's Algorithm mit Capacity Scaling

maxFlow doppelt so schnell wie Ford Fulkerson  $O(|V|^2 \cdot |E|)$   
 addEdge fügt eine gerichtete Kante ein  $O(1)$

```

1 struct Edge {
2     int to, rev;
3     ll f, c;
4 };
5
6 vector<vector<Edge>> adj;
7 int s, t;
8 vector<int> pt, dist;
9
10 void addEdge(int u, int v, ll c) {
11     adj[u].push_back({v, (int)sz(adj[v]), 0, c});
12     adj[v].push_back({u, (int)sz(adj[u]) - 1, 0, 0});
13 }
14
15 bool bfs(ll lim) {
16     dist.assign(sz(adj), -1);
17     dist[s] = 0;
18     queue<int> q({s});
19     while (!q.empty() && dist[t] < 0) {
20         int v = q.front(); q.pop();
21         for (Edge& e : adj[v]) {
22             if (dist[e.to] < 0 && e.c - e.f >= lim) {
23                 dist[e.to] = dist[v] + 1;
24                 q.push(e.to);
25             }
26         }
27     }
28     return dist[t] >= 0;
29 }
30
31 bool dfs(int v, ll flow) {
32     if (v == t) return true;
33     for (; pt[v] < sz(adj[v]); pt[v]++) {
34         Edge& e = adj[v][pt[v]];
35         if (dist[e.to] != dist[v] + 1) continue;
36         if (e.c - e.f >= flow && dfs(e.to, flow)) {
37             e.f += flow;
38             adj[e.to][e.rev].f -= flow;
39             return true;
40         }
41     }
42     return false;
43 }
44
45 ll maxFlow(int source, int target) {
46     s = source, t = target;
47     ll flow = 0;
48     for (ll lim = (1LL << 62); lim >= 1; lim /= 2) {
49         while (bfs(lim)) {
50             pt.assign(sz(adj), 0);
51             while (dfs(s, lim)) flow += lim;
52         }
53     }
54     return flow;
55 }

```

## 2.25 Maximum Weight Bipartite Matching

match berechnet Matching  $O(|V|^3)$

```
1 double costs[N_LEFT][N_RIGHT];
2 // Es muss l<=r sein! (sonst Endlosschleife)
3 double match(int l, int r) {
4     vector<double> lx(l), ly(r);
5     //xy is matching from l->r, yx from r->l, or -1
6     vector<int> xy(l, -1), yx(r, -1);
7     vector<pair<double, int>> slack(r);
8     for (int x = 0; x < l; x++)
9         lx[x] = *max_element(costs[x], costs[x] + r);
10    for (int root = 0; root < l; root++) {
11        vector<int> aug(r, -1);
12        vector<bool> s(l);
13        s[root] = true;
14        for (int y = 0; y < r; y++) {
15            slack[y] = {lx[root] + ly[y] - costs[root][y], root};
16        }
17        int y = -1;
18        while (true) {
19            double delta = INF;
20            int x = -1;
21            for (int yy = 0; yy < r; yy++) {
22                if (aug[yy] < 0 && slack[yy].first < delta) {
23                    tie(delta, x) = slack[yy];
24                    y = yy;
25                }
26            }
27            if (delta > 0) {
28                for (int x = 0; x < l; x++) if (s[x]) lx[x] -= delta;
29                for (int y = 0; y < r; y++) {
30                    if (aug[y] >= 0) ly[y] += delta;
31                    else slack[y].first -= delta;
32                }
33                aug[y] = x;
34                x = yx[y];
35                if (x < 0) break;
36                s[x] = true;
37                for (int y = 0; y < r; y++) {
38                    if (aug[y] < 0) {
39                        double alt = lx[x] + ly[y] - costs[x][y];
40                        if (slack[y].first > alt) {
41                            slack[y] = {alt, x};
42                        }
43                    }
44                }
45                while (y >= 0) {
46                    yx[y] = aug[y];
47                    swap(y, xy[aug[y]]);
48                }
49            }
50            return accumulate(all(lx), 0.0) +
51                accumulate(all(ly), 0.0); // Wert des Matchings
52    }
```

## 3 Geometrie

### 3.1 Closest Pair

shortestDist kürzester Abstand zwischen Punkten  $O(n \log(n))$

```
1 ll rec(vector<pt>::iterator a, int l, int r) {
2     if (r - l < 2) return INF;
3     int m = (l + r) / 2;
4     ll midx = a[m].real();
5     ll ans = min(rec(a, l, m), rec(a, m, r));
6     inplace_merge(a+l, a+m, a+r, [](const pt& x, const pt& y) {
7         return x.imag() < y.imag();
8     });
9     pt tmp[8];
10    fill(all(tmp), a[l]);
11    for (int i = l + 1, next = 0; i < r; i++) {
12        if (ll x = a[i].real() - midx; x * x < ans) {
13            for (pt& p : tmp) ans = min(ans, norm(p - a[i]));
14            tmp[next++ & 7] = a[i];
15        }
16    }
17    return ans;
18 }
19 ll shortestDist(vector<pt> a) { // sz(pts) > 1
20     sort(all(a), [](const pt& x, const pt& y) {
21         return x.real() < y.real();
22     });
23     return rec(a.begin(), 0, sz(a));
24 }
```

### 3.2 Konvexehülle

convexHull berechnet konvexe Hülle  $O(n \log(n))$

- konvexe Hülle gegen den Uhrzeigersinn sortiert
- nur Eckpunkte enthalten (für alle Punkte = im CCW Test entfernen)
- erster und letzter Punkt sind identisch

```
1 vector<pt> convexHull(vector<pt> pts){
2     sort(all(pts), [](const pt& a, const pt& b){
3         return real(a) == real(b) ? imag(a) < imag(b)
4             : real(a) < real(b);
5     });
6     pts.erase(unique(all(pts)), pts.end());
7     int k = 0;
8     vector<pt> h(2 * sz(pts));
9     auto half = [&](auto begin, auto end, int t) {
10        for (auto it = begin; it != end; it++) {
11            while (k > t && cross(h[k-2], h[k-1], *it) <= 0) k--;
12            h[k++] = *it;
13        }
14        half(all(pts), 1); // Untere Hülle.
15        half(next(pts.rbegin()), pts.rend(), k); // Obere Hülle.
16        h.resize(k);
17        return h;
18 }
```

## 3.3 Rotating calipers

antipodalPoints berechnet antipodale Punkte  $O(n)$

WICHTIG: Punkte müssen gegen den Uhrzeigersinn sortiert sein und konvexes Polygon bilden!

```
1 vector<pair<int, int>> antipodalPoints(vector<pt>& h) {
2     if (sz(h) < 2) return {};
3     vector<pair<int, int>> result;
4     for (int i = 0, j = 1; i < j; i++) {
5         while (true) {
6             result.push_back({i, j});
7             if (cross(h[(i + 1) % sz(h)] - h[i],
8                 h[(j + 1) % sz(h)] - h[j]) <= 0) break;
9             j = (j + 1) % sz(h);
10        }
11        result.push_back({i, j});
12    }
```

## 3.4 Formeln – std::complex

```
1 // Komplexe Zahlen als Punkte. Wenn immer möglich complex<ll>
2 // verwenden. Funktionen wie abs() geben dann aber ll zurück.
3 using pt = complex<double>;
4 constexpr double PIU = acos(-1.0l); // PI < PIU
5 constexpr double PIL = PIU-2e-19l;
6 // Winkel zwischen Punkt und x-Achse in [-PI, PI].
7 double angle(pt a) {return arg(a);}
8 // rotiert Punkt im Uhrzeigersinn um den Ursprung.
9 pt rotate(pt a, double theta) {return a * polar(1.0, theta);}
10 // Skalarprodukt.
11 auto dot(pt a, pt b) {return real(conj(a) * b);}
12 // abs()^2. (pre c++20)
13 auto norm(pt a) {return dot(a, a);}
14 // Kreuzprodukt, 0, falls kollinear.
15 auto cross(pt a, pt b) {return imag(conj(a) * b);}
16 auto cross(pt p, pt a, pt b) {return cross(a - p, b - p);}
17 // 1 => c links von a->b
18 // 0 => a, b und c kollinear
19 // -1 => c rechts von a->b
20 int ccw(pt a, pt b, pt c) {
21     auto orien = cross(b - a, c - a);
22     return (orien > EPS) - (orien < -EPS);
23 }
24 // Liegt d in der gleichen Ebene wie a, b, und c?
25 bool isCoplanar(pt a, pt b, pt c, pt d) {
26     return abs((b - a) * (c - a) * (d - a)) < EPS;
27 }
28 // charakterisiert Winkel zwischen Vektoren u und v
29 pt uniqueAngle(pt u, pt v) {
30     pt tmp = v * conj(u);
31     ll g = abs(gcd(real(tmp), imag(tmp)));
32     return tmp / g;
33 }
```

```

1 // Liegt p auf der Geraden a-b? 2d und 3d
2 bool pointOnLine(pt a, pt b, pt p) {
3     return ccw(a, b, p) == 0;
4 }
5 // Test auf Linienschnitt zwischen a-b und c-d. (nicht identisch)
6 bool lineIntersection(pt a, pt b, pt c, pt d) {
7     return abs(cross(a - b, c - d)) < EPS;
8 }
9 // Berechnet den Schnittpunkt der Geraden a-b und c-d.
10 // die Geraden dürfen nicht parallel sein!
11 pt lineIntersection2(pt a, pt b, pt c, pt d) {
12     double x = cross(b - a, d - c);
13     double y = cross(c - a, d - c);
14     return a + y/x*(b - a);
15 }
16 // Entfernung von Punkt p zur Geraden durch a-b. 2d und 3d
17 double distToLine(pt a, pt b, pt p) {
18     return abs(cross(p - a, b - a)) / abs(b - a);
19 }
20 // Projiziert p auf die Gerade a-b
21 pt projectToLine(pt a, pt b, pt p) {
22     return a + (b - a) * dot(p - a, b - a) / norm(b - a);
23 }
24 // sortiert alle Punkte pts auf einer Linie entsprechend dir
25 void sortLine(pt dir, vector<pt>& pts) { // (2d und 3d)
26     sort(all(pts), [&](pt a, pt b){
27         return dot(dir, a) < dot(dir, b);
28     });
29 }
30 // Liegt p auf der Strecke a-b? (nutze < für innerhalb)
31 bool pointOnSegment(pt a, pt b, pt p) {
32     if (ccw(a, b, p) != 0) return false;
33     auto dist = norm(a - b);
34     return norm(a - p) <= dist && norm(b - p) <= dist;
35 }
36 // Entfernung von Punkt p zur Strecke a-b.
37 double distToSegment(pt a, pt b, pt p) {
38     if (a == b) return abs(p - a);
39     if (dot(p - a, b - a) <= 0) return abs(p - a);
40     if (dot(p - b, b - a) >= 0) return abs(p - b);
41     return distToLine(a, b, p);
42 }
43 // Test auf Streckenschnitt zwischen a-b und c-d.
44 bool segmentIntersection(pt a, pt b, pt c, pt d) {
45     if (ccw(a, b, c) == 0 && ccw(a, b, d) == 0)
46         return pointOnSegment(a, b, c) ||
47                pointOnSegment(a, b, d) ||
48                pointOnSegment(c, d, a) ||
49                pointOnSegment(c, d, b);
50     return ccw(a, b, c) * ccw(a, b, d) <= 0 &&
51            ccw(c, d, a) * ccw(c, d, b) <= 0;
52 }
53 // Berechnet die Schnittpunkte der Strecken a-b und c-d.
54 // Enthält entweder keinen Punkt, den einzigen Schnittpunkt

```

```

55 // oder die Endpunkte der Schnittstrecke.
56 vector<pt> segmentIntersection2(pt a, pt b, pt c, pt d) {
57     double x = cross(b - a, d - c);
58     double y = cross(c - a, d - c);
59     double z = cross(b - a, a - c);
60     if (x < 0) {x = -x; y = -y; z = -z;}
61     if (y < -EPS || y-x > EPS || z < -EPS || z-x > EPS) return {};
62     if (x > EPS) return {a + y/x*(b - a)};
63     vector<pt> result;
64     auto insertUnique = [&](pt p) {
65         for (auto q : result) if (abs(p - q) < EPS) return;
66         result.push_back(p);
67     };
68     if (dot(c-a, d-a) < EPS) insertUnique(a);
69     if (dot(c-b, d-b) < EPS) insertUnique(b);
70     if (dot(a-c, b-c) < EPS) insertUnique(c);
71     if (dot(a-d, b-d) < EPS) insertUnique(d);
72     return result;
73 }
74 // Kürzeste Entfernung zwischen den Strecken a-b und c-d.
75 double distBetweenSegments(pt a, pt b, pt c, pt d) {
76     if (segmentIntersection(a, b, c, d)) return 0.0;
77     return min({distToSegment(a, b, c), distToSegment(a, b, d),
78                distToSegment(c, d, a), distToSegment(c, d, b)});
79 }

```

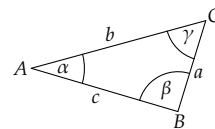
```

1 bool left(pt p) {return real(p) < 0 ||
2     (real(p) == 0 && imag(p) < 0);}
3 // counter clockwise, starting with "11:59"
4 void sortAround(pt p, vector<pt>& ps) {
5     sort(all(ps), [&](const pt& a, const pt& b){
6         if (left(a - p) != left(b - p))
7             return left(a - p) > left(b - p);
8         return cross(p, a, b) > 0;
9     });
10 }

```

Generell:

$$\begin{aligned} \cos(\gamma) &= \frac{a^2 + b^2 - c^2}{2ab} \\ b &= \frac{a}{\sin(\alpha)} \sin(\beta) \\ \Delta &= \frac{bc}{2} \sin(\alpha) \end{aligned}$$

 $\beta = 90^\circ$ :

$$\begin{aligned} \sin(\alpha) &= \frac{a}{b} \\ \cos(\alpha) &= \frac{c}{b} \\ \tan(\alpha) &= \frac{a}{c} \end{aligned}$$

```

1 // Mittelpunkt des Dreiecks abc.
2 pt centroid(pt a, pt b, pt c) {return (a + b + c) / 3.0;}
3 // Flächeninhalt eines Dreiecks bei bekannten Eckpunkten.
4 double area(pt a, pt b, pt c) {
5     return abs(cross(a, b, c)) / 2.0;
6 }
7 // Flächeninhalt eines Dreiecks bei bekannten Seitenlängen.
8 double area(double a, double b, double c) {
9     double s = (a + b + c) / 2.0; //unpräzise
10    return sqrt(s * (s-a) * (s-b) * (s-c));
11 }
12 // Zentrum des größten Kreises im Dreieck
13 pt inCenter(pt a, pt b, pt c) {

```

```

14 double x = abs(a-b), y = abs(b-c), z = abs(a-c);
15 return (y*a + z*b + x*c) / (x+y+z);
16 }
17 // Zentrum des Kreises durch alle Eckpunkte
18 // a, b und c nicht kollinear
19 pt circumCenter(pt a, pt b, pt c) {
20     b -= a, c -= a;
21     pt d = b * norm(c) - c * norm(b);
22     d = {-d.imag(), d.real()};
23     return a + d / cross(b, c) / 2.0;
24 }
25 // -1 => p außerhalb Kreis durch a,b,c
26 // 0 => p auf Kreis durch a,b,c
27 // 1 => p im Kreis durch a,b,c
28 int insideOutCenter(pt a, pt b, pt c, pt p) { // braucht lll
29     return ccw(a,b,c) * sgn(imag((c-b)*conj(p-c)*(a-p)*conj(b-a)));
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 verschiedener Orientierung
34 bool similar(pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
35     return (b2-a2) * (c1-a1) == (b1-a1) * (c2-a2) ||
36            (b2-a2) * conj(c1-a1) == conj(b1-a1) * (c2-a2);
37 }

```

```

1 // Flächeninhalt eines Polygons (nicht selbstschneidend).
2 // Punkte gegen den Uhrzeigersinn: positiv, sonst negativ.
3 double area(const vector<pt>& poly) { //poly[0] == poly.back()
4     ll res = 0;
5     for (int i = 0; i + 1 < sz(poly); i++)
6         res += cross(poly[i], poly[i + 1]);
7     return 0.5 * res;
8 }
9 // Anzahl ccw drehungen einer Polyline um einen Punkt
10 // p nicht auf rand und poly[0] == poly.back()
11 // res != 0 or (res & 1) != 0 um inside zu prüfen bei
12 // selbstschneidenden Polygonen (definitions Sache)
13 ll windingNumber(pt p, const vector<pt>& poly) {
14     ll res = 0;
15     for (int i = 0; i + 1 < sz(poly); i++) {
16         pt a = poly[i], b = poly[i + 1];
17         if (real(a) > real(b)) swap(a, b);
18         if (real(a) <= real(p) && real(p) < real(b) &&
19             cross(p, a, b) < 0) {
20             res += ccw(p, poly[i], poly[i + 1]);
21         }
22     }
23     return res;
24 }
25 // Testet, ob ein Punkt im Polygon liegt (beliebige Polygone).
26 // Ändere Zeile 32 falls rand zählt, poly[0] == poly.back()
27 bool inside(pt p, const vector<pt>& poly) {
28     bool in = false;
29     for (int i = 0; i + 1 < sz(poly); i++) {
30         pt a = poly[i], b = poly[i + 1];
31         if (pointOnLineSegment(a, b, p)) return false;

```

```

31 if (real(a) > real(b)) swap(a,b);
32 if (real(a) <= real(p) && real(p) < real(b) &&
33     cross(p, a, b) < 0) {
34     in ^= 1;
35 }
36 return in;
37 }
38 // convex hull without duplicates, h[0] != h.back()
39 // apply comments if border counts as inside
40 bool insideConvex(pt p, const vector<pt>& hull) {
41     int l = 0, r = sz(hull) - 1;
42     if (cross(hull[l], hull[r], p) >= 0) return false; // > 0
43     while (l + 1 < r) {
44         int m = (l + r) / 2;
45         if (cross(hull[l], hull[m], p) > 0) l = m; // >= 0
46         else r = m;
47     }
48     return cross(hull[l], hull[r], p) > 0; // >= 0
49 }
50 void rotateMin(vector<pt>& hull) {
51     auto mi = min_element(all(hull), [](const pt& a, const pt& b) {
52         return real(a) == real(b) ? imag(a) < imag(b)
53             : real(a) < real(b);
54     });
55     rotate(hull.begin(), mi, hull.end());
56 }
57 // convex hulls without duplicates, h[0] != h.back()
58 vector<pt> minkowski(vector<pt> ps, vector<pt> qs) {
59     rotateMin(ps);
60     rotateMin(qs);
61     ps.push_back(ps[0]);
62     qs.push_back(qs[0]);
63     ps.push_back(ps[1]);
64     qs.push_back(qs[1]);
65     vector<pt> res;
66     for (ll i = 0, j = 0; i + 2 < sz(ps) || j + 2 < sz(qs);) {
67         res.push_back(ps[i] + qs[j]);
68         auto c = cross(ps[i + 1] - ps[i], qs[j + 1] - qs[j]);
69         if (c >= 0) i++;
70         if (c <= 0) j++;
71     }
72     return res;
73 }
74 // convex hulls without duplicates, h[0] != h.back()
75 double dist(const vector<pt>& ps, vector<pt> qs) {
76     for (pt& q : qs) q *= -1;
77     auto p = minkowski(ps, qs);
78     p.push_back(p[0]);
79     double res = INF;
80     bool intersect = true;
81     for (ll i = 0; i + 1 < sz(p); i++) {
82         intersect &= cross(p[i], p[i+1]) >= 0;
83         res = min(res, distToSegment(p[i], p[i+1], 0));
84     }
85     return intersect ? 0 : res;
86 }

```

```

87 bool left(pt of, pt p) {return cross(p, of) < 0 ||
88     (cross(p, of) == 0 && dot(p, of) > 0);}
89 // convex hulls without duplicates, hull[0] == hull.back() and
90 // hull[0] must be a convex point (with angle < pi)
91 // returns index of corner where dot(dir, corner) is maximized
92 int extremal(const vector<pt>& hull, pt dir) {
93     dir *= pt(0, 1);
94     int l = 0, r = sz(hull) - 1;
95     while (l + 1 < r) {
96         int m = (l + r) / 2;
97         pt dm = hull[m+1] - hull[m];
98         pt dl = hull[l+1] - hull[l];
99         if (left(dl, dir) != left(dl, dm)) {
100             if (left(dl, dm)) l = m;
101             else r = m;
102         } else {
103             if (cross(dir, dm) < 0) l = m;
104             else r = m;
105         }
106     }
107     return r % (sz(hull) - 1);
108 }
109 // convex hulls without duplicates, hull[0] == hull.back() and
110 // hull[0] must be a convex point (with angle < pi)
111 // {} if no intersection
112 // {x} if corner is only intersection
113 // {i, j} segments (i,i+1) and (j,j+1) intersected (if only the
114 // border is intersected corners i and j are the start and end)
115 vector<int> intersectLine(const vector<pt>& hull, pt a, pt b) {
116     int endA = extremal(hull, (a-b) * pt(0, 1));
117     int endB = extremal(hull, (b-a) * pt(0, 1));
118     // cross == 0 => line only intersects border
119     if (cross(hull[endA], a, b) > 0 ||
120         cross(hull[endB], a, b) < 0) return {};
121     int n = sz(hull) - 1;
122     vector<int> res;
123     for (auto _ : {0, 1}) {
124         int l = endA, r = endB;
125         if (r < l) r += n;
126         while (l + 1 < r) {
127             int m = (l + r) / 2;
128             if (cross(hull[m % n], a, b) <= 0 &&
129                 cross(hull[m % n], a, b) != cross(hull[endB], a, b))
130                 l = m;
131             else r = m;
132         }
133         if (cross(hull[r % n], a, b) == 0) l++;
134         res.push_back(l % n);
135         swap(endA, endB);
136         swap(a, b);
137     }
138     if (res[0] == res[1]) res.pop_back();
139     return res;
140 }
141 // berechnet die Schnittpunkte von zwei Kreisen
142 // (Kreise dürfen nicht gleich sein!)

```

```

3 vector<pt> circleIntersection(pt c1, double r1,
4                               pt c2, double r2) {
5     double d = abs(c1 - c2);
6     if (d < abs(r1 - r2) || d > abs(r1 + r2)) return {};
7     double a = (r1 * r1 - r2 * r2 + d * d) / (2 * d);
8     pt p = (c2 - c1) * a / d + c1;
9     if (d == abs(r1 - r2) || d == abs(r1 + r2)) return {p};
10    double h = sqrt(r1 * r1 - a * a);
11    return {p + pt(0, 1) * (c2 - c1) * h / d,
12        p - pt(0, 1) * (c2 - c1) * h / d};
13 }
14 // berechnet die Schnittpunkte zwischen
15 // einem Kreis(Kugel) und einem Strahl (2D und 3D)
16 vector<pt> circleRayIntersection(pt center, double r,
17                                  pt orig, pt dir) {
18     vector<pt> result;
19     double a = norm(dir);
20     double b = 2 * dot(dir, orig - center);
21     double c = norm(orig - center) - r * r;
22     double discr = b * b - 4 * a * c;
23     if (discr >= 0) {
24         //t in [0, 1] => schnitt mit Segment [orig, orig + dir]
25         double t1 = -(b + sqrt(discr)) / (2 * a);
26         double t2 = -(b - sqrt(discr)) / (2 * a);
27         if (t1 >= 0) result.push_back(t1 * dir + orig);
28         if (t2 >= 0 && abs(t1 - t2) > EPS) {
29             result.push_back(t2 * dir + orig);
30         }
31     }
32     return result;
33 }

```

### 3.5 Formeln – 3D

```

1 // Skalarprodukt
2 auto operator|(pt3 a, pt3 b) {
3     return a.x * b.x + a.y * b.y + a.z * b.z;
4 }
5 auto dot(pt3 a, pt3 b) {return a|b;}
6 // Kreuzprodukt
7 pt3 operator*(pt3 a, pt3 b) {return {a.y*b.z - a.z*b.y,
8     a.z*b.x - a.x*b.z,
9     a.x*b.y - a.y*b.x};}
10 pt3 cross(pt3 a, pt3 b) {return a*b;}
11 // Länge von a
12 double abs(pt3 a) {return sqrt(dot(a, a));}
13 double abs(pt3 a, pt3 b) {return abs(b - a);}
14 // Mixedprodukt
15 auto mixed(pt3 a, pt3 b, pt3 c) {return a*b|c;}
16 // Orientierung von p zu der Ebene durch a, b, c
17 // -1 => gegen den Uhrzeigersinn,
18 // 0 => kollinear,
19 // 1 => im Uhrzeigersinn.
20 int ccw(pt3 a, pt3 b, pt3 c, pt3 p) {
21     auto orien = mixed(b - a, c - a, p - a);
22     return (orien > EPS) - (orien < -EPS);
23 }

```



```

24 // Entfernung von Punkt p zur Ebene a,b,c.
25 double distToPlane(pt3 a, pt3 b, pt3 c, pt3 p) {
26     pt3 n = cross(b-a, c-a);
27     return (abs(dot(n, p)) - dot(n, a)) / abs(n);
28 }
29 // Liegt p in der Ebene a,b,c?
30 bool pointOnPlane(pt3 a, pt3 b, pt3 c, pt3 p) {
31     return ccw(a, b, c, p) == 0;
32 }
33 // Schnittpunkt von der Grade a-b und der Ebene c,d,e
34 // die Grade darf nicht parallel zu der Ebene sein!
35 pt3 linePlaneIntersection(pt3 a, pt3 b, pt3 c, pt3 d, pt3 e) {
36     pt3 n = cross(d-c, e-c);
37     pt3 d = b - a;
38     return a - d * (dot(n, a) - dot(n, c)) / dot(n, d);
39 }
40 // Abstand zwischen der Grade a-b und c-d
41 double lineLineDist(pt3 a, pt3 b, pt3 c, pt3 d) {
42     pt3 n = cross(b - a, d - c);
43     if (abs(n) < EPS) return distToLine(a, b, c);
44     return abs(dot(a - c, n)) / abs(n);
45 }

```

### 3.6 Half-plane intersection

```

1 constexpr ll inf = 0x1FFF'FFFF'FFFF'FFFF; //THIS CODE IS WIP
2 bool left(pt p) {return real(p) < 0 ||
3     (real(p) == 0 && imag(p) < 0);}
4 struct hp {
5     pt from, to;
6     hp(pt a, pt b) : from(a), to(b) {}
7     hp(pt dummy) : hp(dummy, dummy) {}
8     bool dummy() const {return from == to;}
9     pt dir() const {return dummy() ? to : to - from;}
10    bool operator<(const hp& o) const {
11        if (left(dir()) != left(o.dir()))
12            return left(dir()) > left(o.dir());
13        return cross(dir(), o.dir()) > 0;
14    }
15    using lll = __int128;
16    using ptl = complex<lll>;
17    ptl mul(lll m, ptl p) const {return m*p;} //ensure 128bit
18    bool check(const hp& a, const hp& b) const {
19        if (dummy() || b.dummy()) return false;
20        if (a.dummy()) {
21            ll ort = sgn(cross(b.dir(), dir()));
22            if (ort == 0) return cross(from, to, a.from) < 0;
23            return cross(b.dir(), a.dir()) * ort > 0;
24        }
25        ll y = cross(a.dir(), b.dir());
26        ll z = cross(b.from - a.from, b.dir());
27        ptl i = mul(y, a.from) + mul(z, a.dir()); //intersect a and b
28        // check if i is outside/right of x
29        return imag(conj(mul(sgn(y), dir()))*(i-mul(y, from))) < 0;
30    }

```

```

31 };
32 constexpr ll lim = 2e9+7;
33 deque<hp> intersect(vector<hp> hps) {
34     hps.push_back(hp(pt{lim+1, -1}));
35     hps.push_back(hp(pt{lim+1, 1}));
36     sort(all(hps));
37     deque<hp> dq = {hp(pt{-lim, 1})};
38     for (auto x : hps) {
39         while (sz(dq) > 1 && x.check(dq.end()[-1], dq.end()[-2]))
40             dq.pop_back();
41         while (sz(dq) > 1 && x.check(dq[0], dq[1]))
42             dq.pop_front();
43         if (cross(x.dir(), dq.back().dir()) == 0) {
44             if (dot(x.dir(), dq.back().dir()) < 0) return {};
45             if (cross(x.from, x.to, dq.back().from) < 0)
46                 dq.pop_back();
47             else continue;
48         }
49         dq.push_back(x);
50     }
51     while (sz(dq) > 2 && dq[0].check(dq.end()[-1], dq.end()[-2]))
52         dq.pop_back();
53     while (sz(dq) > 2 && dq.end()[-1].check(dq[0], dq[1]))
54         dq.pop_front();
55     if (sz(dq) < 3) return {};
56     return dq;
57 }

```

## 4 Mathe

### 4.1 Longest Increasing Subsequence

- lower\_bound  $\Rightarrow$  streng monoton
- upper\_bound  $\Rightarrow$  monoton

```

1 vector<int> lis(vector<ll>& a) {
2     int n = sz(a), len = 0;
3     vector<ll> dp(n, INF), dp_id(n), prev(n);
4     for (int i = 0; i < n; i++) {
5         int pos = lower_bound(all(dp), a[i]) - dp.begin();
6         dp[pos] = a[i];
7         dp_id[pos] = i;
8         prev[i] = pos ? dp_id[pos - 1] : -1;
9         len = max(len, pos + 1);
10    }
11    // reconstruction
12    vector<int> res(len);
13    for (int x = dp_id[len-1]; len--; x = prev[x]) {
14        res[len] = x;
15    }
16    return res; // indices of one LIS
17 }

```

### 4.2 Zykel Erkennung

cycleDetection findet Zyklus von  $x_0$  und Länge in  $f$   $O(b+l)$

```

1 pair<ll, ll> cycleDetection(ll x0, function<ll(ll)> f) {
2     ll a = x0, b = f(x0), length = 1;
3     for (ll power = 1; a != b; b = f(b), length++) {
4         if (power == length) {
5             power *= 2;
6             length = 0;
7             a = b;
8         }
9         ll start = 0;
10        a = x0; b = x0;
11        for (ll i = 0; i < length; i++) b = f(b);
12        while (a != b) {
13            a = f(a);
14            b = f(b);
15            start++;
16        }
17        return {start, length};
18    }

```

### 4.3 Permutationen

kthperm findet  $k$ -te Permutation ( $k \in [0, n!)$ )  $O(n \cdot \log(n))$

```

1 vector<ll> kthperm(ll n, ll k) {
2     Tree<ll> t;
3     vector<ll> res(n);
4     for (ll i = 1; i <= n; k /= i, i++) {
5         t.insert(i - 1);
6         res[n - i] = k % i;
7     }
8     for (ll& x : res) {
9         auto it = t.find_by_order(x);
10        x = *it;
11        t.erase(it);
12    }
13    return res;
14 }

```

permIndex bestimmt Index der Permutation ( $res \in [0, n!)$ )  $O(n \cdot \log(n))$

```

1 ll permIndex(vector<ll> v) {
2     Tree<ll> t;
3     reverse(all(v));
4     for (ll& x : v) {
5         t.insert(x);
6         x = t.order_of_key(x);
7     }
8     ll res = 0;
9     for (int i = sz(v); i > 0; i--) {
10        res = res * i + v[i - 1];
11    }
12    return res;
13 }

```

#### 4.4 Mod-Exponent und Multiplikation über $\mathbb{F}_p$

mulMod berechnet  $a \cdot b \bmod n$   $O(\log(b))$

```
1 ll mulMod(ll a, ll b, ll n) {
2     ll res = 0;
3     while (b > 0) {
4         if (b & 1) res = (a + res) % n;
5         a = (a * 2) % n;
6         b /= 2;
7     }
8     return res;
9 }
```

powMod berechnet  $a^b \bmod n$   $O(\log(b))$

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

- für  $a > 10^9$  `__int128` oder `modMul` benutzen!

#### 4.5 ggT, kgV, erweiterter euklidischer Algorithmus

$O(\log(a) + \log(b))$

```
1 // a*x + b*y = ggT(a, b)
2 array<ll, 3> extendedEuclid(ll a, ll b) {
3     if (a == 0) return {b, 0, 1};
4     auto [d, x, y] = extendedEuclid(b % a, a);
5     return {d, y - (b / a) * x, x};
6 }
```

#### 4.6 Multiplikatives Inverses von $x$ in $\mathbb{Z}/m\mathbb{Z}$

Falls  $m$  prim:  $x^{-1} \equiv x^{m-2} \bmod m$

Falls  $\text{ggT}(x, m) = 1$ :

- Erweiterter euklidischer Algorithmus liefert  $\alpha$  und  $\beta$  mit  $\alpha x + \beta m = 1$ .
- Nach Kongruenz gilt  $\alpha x + \beta m \equiv \alpha x \equiv 1 \bmod m$ .
- $x^{-1} \equiv \alpha \bmod m$

Sonst  $\text{ggT}(x, m) > 1$ : Es existiert kein  $x^{-1}$ .

```
1 ll multInv(ll x, ll m) { // x^{-1} mod m
2     return 1 < x ? m - multInv(m % x, x) * m / x : 1;
3 }
```

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

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

**PELL-Gleichungen** Sei  $(\bar{x}, \bar{y})$  die Lösung von  $x^2 - ny^2 = 1$ , die  $x > 1$  minimiert. Sei  $(\tilde{x}, \tilde{y})$  die Lösung von  $x^2 - ny^2 = c$ , die  $x > 1$  minimiert. Dann lassen sich alle Lösungen von  $x^2 - ny^2 = c$  berechnen durch:

$$\begin{aligned} x_1 &:= \bar{x}, & y_1 &:= \bar{y} \\ x_{k+1} &:= \bar{x}x_k + n\bar{y}y_k, & y_{k+1} &:= \bar{x}y_k + \bar{y}x_k \end{aligned}$$

#### 4.7 Lineare Kongruenz

- Kleinste Lösung  $x$  für  $ax \equiv b \pmod{m}$ .
- Weitere Lösungen unterscheiden sich um  $\frac{m}{g}$ , es gibt also  $g$  Lösungen modulo  $m$ .

```
1 ll solveLinearCongruence(ll a, ll b, ll m) {
2     ll g = gcd(a, m);
3     if (b % g != 0) return -1;
4     return ((b / g) * multInv(a / g, m / g)) % (m / g);
5 }
```

#### 4.8 Chinesischer Restsatz

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

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

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

```
1 struct CRT {
2     using ll = __int128;
3     ll M = 1, sol = 0; // Solution unique modulo M
4     bool hasSol = true;
5     // Adds congruence x = a (mod m)
6     void add(ll a, ll m) {
7         auto [d, s, t] = extendedEuclid(M, m);
8         if ((a - sol) % d != 0) hasSol = false;
9         ll z = M/d * s;
10        M *= m/d;
11        sol = (z % M * (a - sol) % M + sol + M) % M;
12    }
13    };
```

#### 4.9 Primzahltest & Faktorisierung

isPrime prüft ob Zahl prim ist  $O(\log(n)^2)$

```
1 constexpr ll bases32[] = {2, 7, 61};
2 constexpr ll bases64[] = {2, 325, 9375, 28178, 450775,
3     9780504, 1795265022};
4 bool isPrime(ll n) {
5     if (n < 2 || n % 2 == 0) return n == 2;
6     ll d = n - 1, j = 0;
7     while (d % 2 == 0) d /= 2, j++;
8     for (ll a : bases64) {
9         if (a % n == 0) continue;
10        ll v = powMod(a, d, n); //with mulmod or int128
11        if (v == 1 || v == n - 1) continue;
12        for (ll i = 1; i <= j; i++) {
13            v = ((ll)v * v) % n;
14            if (v == n - 1 || v == 1) break;
15        }
16        if (v != n - 1) return false;
17    }
18    return true;
19 }
```

rho findet zufälligen Teiler  $O(\sqrt[4]{n})$

```
1 using ll = __int128;
2 ll rho(ll n) { // Findet Faktor < n, nicht unbedingt prim.
3     if (n % 2 == 0) return 2;
4     ll x = 0, y = 0, prd = 2, i = n/2 + 7;
5     auto f = [&](ll c){return (c * c + i) % n;};
6     for (ll t = 30; t % 40 || gcd(prd, n) == 1; t++) {
7         if (x == y) x = ++i, y = f(x);
8         if (ll q = (ll)prd * abs(x-y) % n; q) prd = q;
9         x = f(x); y = f(f(y));
10    }
11    return gcd(prd, n);
12 }
13 void factor(ll n, map<ll, int>& facts) {
14     if (n == 1) return;
15     if (isPrime(n)) {facts[n]++; return;}
16     ll f = rho(n);
17     factor(n / f, facts); factor(f, facts);
18 }
```

#### 4.10 Teiler

countDivisors Zählt Teiler von  $n$   $O(\sqrt[3]{n})$

```
1 ll countDivisors(ll n) {
2     ll res = 1;
3     for (ll i = 2; i * i * i <= n; i++) {
4         ll c = 0;
5         while (n % i == 0) {n /= i; c++;}
6         res *= c + 1;
7     }
8     if (isPrime(n)) res *= 2;
9     else if (n > 1) res *= isSquare(n) ? 3 : 4;
10    return res;
11 }
```

#### 4.11 Matrix-Exponent

precalc berechnet  $m^{2^b}$  vor  $O(\log(b) \cdot n^3)$   
calc berechnet  $m^b$ .  $O(\log(b) \cdot n^2)$

Tipp: wenn  $v[x]=1$  und 0 sonst, dann ist  $\text{res}[y] = m_{y,x}^b$ .

```
1 vector<mat> pows;
2 void precalc(mat m) {
3     pows = {mat(sz(m.m), 1), m};
4     for (int i = 1; i < 60; i++) pows.push_back(pows[i] * pows[i]);
5 }
6 auto calc(ll b, vector<ll> v) {
7     for (ll i = 1; b > 0; i++) {
8         if (b & 1) v = pows[i] * v;
9         b /= 2;
10    }
11    return v;
12 }
```

## 4.12 Lineare Rekurrenz

BerlekampMassey Berechnet eine lineare Rekurrenz  $n$ -ter Ordnung  $O(n^2)$  aus den ersten  $2n$  Werte

```

1 constexpr ll mod = 1'000'000'007;
2 vector<ll> BerlekampMassey(const vector<ll>& s) {
3     int n = sz(s), L = 0, m = 0;
4     vector<ll> C(n), B(n), T;
5     C[0] = B[0] = 1;
6     ll b = 1;
7     for (int i = 0; i < n; i++) {
8         m++;
9         ll d = s[i] % mod;
10        for (int j = 1; j <= L; j++) {
11            d = (d + C[j] * s[i - j]) % mod;
12        }
13        if (!d) continue;
14        T = C;
15        ll coef = d * powMod(b, mod-2, mod) % mod;
16        for (int j = m; j < n; j++) {
17            C[j] = (C[j] - coef * B[j - m]) % mod;
18        }
19        if (2 * L > i) continue;
20        L = i + 1 - L;
21        swap(B, T);
22        b = d;
23        m = 0;
24    }
25    C.resize(L + 1);
26    C.erase(C.begin());
27    for (auto& x : C) x = (mod - x) % mod;
28    return C;
29 }

```

Sei  $f(n) = c_0 f(n-1) + c_1 f(n-2) + \dots + c_{n-1} f(0)$  eine lineare Rekurrenz.  
kthTerm Berechnet  $k$ -ten Term einer Rekurrenz  $n$ -ter Ordnung  $O(\log(k) \cdot n^2)$

```

1 constexpr ll mod = 1'000'000'007;
2 vector<ll> modMul(const vector<ll>& a, const vector<ll>& b,
3                 const vector<ll>& c) {
4     ll n = sz(c);
5     vector<ll> res(n * 2 + 1);
6     for (int i = 0; i <= n; i++) { //a*b
7         for (int j = 0; j <= n; j++) {
8             res[i + j] += a[i] * b[j];
9             res[i + j] %= mod;
10        }
11    }
12    for (int i = 2 * n; i > n; i--) { //res*c
13        for (int j = 0; j < n; j++) {
14            res[i - 1 - j] += res[i] * c[j];
15            res[i - 1 - j] %= mod;
16        }
17    }
18    res.resize(n + 1);
19    return res;
20 }
21 ll kthTerm(const vector<ll>& f, const vector<ll>& c, ll k) {
22     assert(sz(f) == sz(c));
23     vector<ll> tmp(sz(c) + 1, a(sz(c) + 1);

```

```

22 tmp[0] = a[1] = 1; //tmp = (x^k) % c
23 for (k++; k > 0; k /= 2) {
24     if (k & 1) tmp = modMul(tmp, a, c);
25     a = modMul(a, a, c);
26 }
27 ll res = 0;
28 for (int i = 0; i < sz(c); i++) res += (tmp[i+1] * f[i]) % mod;
29 return res % mod;
30 }

```

Alternativ kann der  $k$ -te Term in  $O(n^3 \log(k))$  berechnet werden:

$$\begin{pmatrix} c_0 & c_1 & \dots & \dots & c_{n-1} \\ 1 & 0 & \dots & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} f(n-1) \\ f(n-2) \\ \vdots \\ \vdots \\ f(0) \end{pmatrix} = \begin{pmatrix} f(n-1+k) \\ f(n-2+k) \\ \vdots \\ \vdots \\ f(k) \end{pmatrix} \leftarrow$$

## 4.13 Diskreter Logarithmus

solve bestimmt Lösung  $x$  für  $a^x = b \bmod m$   $O(\sqrt{m} \cdot \log(m))$

```

1 ll dlog(ll a, ll b, ll m) { //a > 0!
2     ll bound = sqrtl(m) + 1; //memory usage bound < p
3     vector<pair<ll, ll>> vals(bound);
4     for (ll i = 0, e = 1; i < bound; i++, e = (e * a) % m) {
5         vals[i] = {e, i};
6     }
7     vals.emplace_back(m, 0);
8     sort(all(vals));
9     ll fact = powMod(a, m - bound - 1, m);
10    for (ll i = 0; i < m; i += bound, b = (b * fact) % m) {
11        auto it = lower_bound(all(vals), pair<ll, ll>{b, 0});
12        if (it->first == b) {
13            return (i + it->second) % m;
14        }
15    }
16    return -1;
17 }

```

## 4.14 Diskrete Quadratwurzel

sqrtMod bestimmt Lösung  $x$  für  $x^2 = a \bmod p$   $O(\log(p))$

Wichtig:  $p$  muss prim sein!

```

1 ll sqrtMod(ll a, ll p) { // teste mit legendre ob lösung existiert
2     if (a < 2) return a;
3     ll t = 0;
4     while (legendre((t*t-4*a) % p, p) >= 0) t = rng() % p;
5     ll b = -t, c = -t, d = 1, m = p;
6     for (m++; m /= 2; b = (a+a-b*b) % p, a = (a*a) % p) {
7         if (m % 2) {
8             d = (c-d*b) % p;
9             c = (c*a) % p;
10        } else {
11            c = (d*a - c*b) % p;
12        }
13    }
14    return (d + p) % p;
15 }

```

## 4.15 Primitivwurzeln

- Primitivwurzel modulo  $n$  existiert  $\Leftrightarrow n \in \{2, 4, p^\alpha, 2 \cdot p^\alpha \mid 2 < p \in \mathbb{P}, \alpha \in \mathbb{N}\}$
- es existiert entweder keine oder  $\varphi(\varphi(n))$  inkongruente Primitivwurzeln
- Sei  $g$  Primitivwurzel modulo  $n$ . Dann gilt:  
Das kleinste  $k$ , sodass  $g^k \equiv 1 \bmod n$ , ist  $k = \varphi(n)$ .

isPrimitive prüft ob  $g$  eine Primitivwurzel ist  $O(\log(\varphi(n)) \cdot \log(n))$   
findPrimitive findet Primitivwurzel (oder -1)  $O(|ans| \cdot \log(\varphi(n)) \cdot \log(n))$

```

1 bool isPrimitive(ll g, ll n, ll phi, map<ll, int>& phiFacts) {
2     if (g == 1) return n == 2;
3     if (gcd(g, n) > 1) return false;
4     for (auto [f, _] : phiFacts)
5         if (powMod(g, phi / f, n) == 1) return false;
6     return true;
7 }
8 bool isPrimitive(ll g, ll n) {
9     ll phin = phi(n); //isPrime(n) => phi(n) = n - 1
10    map<ll, int> phiFacts;
11    factor(phin, phiFacts);
12    return isPrimitive(g, n, phin, phiFacts);
13 }
14 ll findPrimitive(ll n) { //test auf existens geht schneller
15     ll phin = phi(n); //isPrime(n) => phi(n) = n - 1
16     map<ll, int> phiFacts;
17     factor(phin, phiFacts);
18     for (ll res = 1; res < n; res++) // oder zufällige Reihenfolge
19         if (isPrimitive(res, n, phin, phiFacts)) return res;
20     return -1;
21 }

```

## 4.16 Diskrete $n$ -te Wurzel

root bestimmt Lösung  $x$  für  $x^n = b \bmod m$   $O(\sqrt{m} \cdot \log(m))$

Alle Lösungen haben die Form  $g^{c + \frac{i \cdot \varphi(n)}{\gcd(a, \varphi(n))}}$

```

1 ll root(ll a, ll b, ll m) { // a > 0!
2     ll g = findPrimitive(m);
3     ll c = dlog(powMod(g, a, m), b, m);
4     return c < 0 ? -1 : powMod(g, c, m);
5 }

```

## 4.17 LEGENDRE-Symbol

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

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

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

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

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

```

1 ll legendre(ll a, ll p) { // p prim >= 2
2     ll s = powMod(a, p / 2, p);
3     return s < 2 ? s : -1ll;
4 }

```

#### 4.18 Lineares Sieb und Multiplikative Funktionen

Eine (zahlentheoretische) Funktion  $f$  heißt multiplikativ wenn  $f(1) = 1$  und  $f(a \cdot b) = f(a) \cdot f(b)$ , falls  $\text{ggT}(a, b) = 1$ .

⇒ Es ist ausreichend  $f(p^k)$  für alle primen  $p$  und alle  $k$  zu kennen.

sieve berechnet Primzahlen und co.  $O(N)$

sieved Wert der entsprechenden multiplikativen Funktion  $O(1)$

naive Wert der entsprechenden multiplikativen Funktion  $O(\sqrt{n})$

Wichtig: Sieb rechts ist schneller für isPrime oder primes!

```

1 constexpr ll N = 10'000'000;
2 ll small[N], power[N], sieved[N];
3 vector<ll> primes;
4 //wird aufgerufen mit (p^k, p, k) für prime p und k > 0
5 ll mu(ll pk, ll p, ll k) {return -(k == 1);}
6 ll phi(ll pk, ll p, ll k) {return pk - pk / p;}
7 ll div(ll pk, ll p, ll k) {return k+1;}
8 ll divSum(ll pk, ll p, ll k) {return (pk*p-1) / (p - 1);}
9 ll square(ll pk, ll p, ll k) {return k % 2 ? pk / p : pk;}
10 ll squareFree(ll pk, ll p, ll k) {return p;}
11 void sieve() { // O(N)
12     small[1] = power[1] = sieved[1] = 1;
13     for (ll i = 2; i < N; i++) {
14         if (small[i] == 0) {
15             primes.push_back(i);
16             for (ll pk = i, k = 1; pk < N; pk *= i, k++) {
17                 small[pk] = i;
18                 power[pk] = pk;
19                 sieved[pk] = mu(pk, i, k); // Aufruf ändern!
20             }
21             for (ll j=0; i*primes[j] < N && primes[j] < small[i]; j++) {
22                 ll k = i * primes[j];
23                 small[k] = power[k] = primes[j];
24                 sieved[k] = sieved[i] * sieved[primes[j]];
25             }
26             if (i * small[i] < N && power[i] != i) {
27                 ll k = i * small[i];
28                 small[k] = small[i];
29                 power[k] = power[i] * small[i];
30                 sieved[k] = sieved[power[k]] * sieved[k / power[k]];
31             }
32         }
33     }
34     ll naive(ll n) { // O(sqrt(n))
35         ll res = 1;
36         for (ll p = 2; p * p <= n; p++) {
37             if (n % p == 0) {
38                 ll pk = 1;
39                 ll k = 0;
40                 do {
41                     n /= p;
42                     pk *= p;
43                     k++;
44                 } while (n % p == 0);
45                 res *= mu(pk, p, k); // Aufruf ändern!
46             }
47         }
48         if (n > 1) res *= mu(n, n, 1);
49         return res;
50     }
51 }

```

Möbius-Funktion:

- $\mu(n) = +1$ , falls  $n$  quadratfrei ist und gerade viele Primteiler hat
- $\mu(n) = -1$ , falls  $n$  quadratfrei ist und ungerade viele Primteiler hat
- $\mu(n) = 0$ , falls  $n$  nicht quadratfrei ist

EULERSCHE  $\varphi$ -FUNKTION:

- Zählt die relativ primen Zahlen  $\leq n$ .
- $p$  prim,  $k \in \mathbb{N}$ :  $\varphi(p^k) = p^k - p^{k-1}$
- Euler's Theorem: Für  $b \geq \varphi(c)$  gilt:  $a^b \equiv a^{b \bmod \varphi(c) + \varphi(c)} \pmod{c}$ . Darüber hinaus gilt:  $\text{gcd}(a, c) = 1 \Leftrightarrow a^b \equiv a^{b \bmod \varphi(c)} \pmod{c}$ . Falls  $m$  prim ist, liefert das den kleinen Satz von FERMAT:  $a^m \equiv a \pmod{m}$

#### 4.19 Primzahlsieb von ERATOSTHENES

- Bis  $10^8$  in unter 64MB Speicher (lange Berechnung)

primeSieve berechnet Primzahlen und Anzahl  $O(N \cdot \log(\log(N)))$

isPrime prüft ob Zahl prim ist  $O(1)$

```

1 constexpr ll N = 100'000'000;
2 bitset<N / 2> isNotPrime;
3 vector<ll> primes = {2};
4 bool isPrime(ll x) {
5     if (x < 2 || x % 2 == 0) return x == 2;
6     else return !isNotPrime[x / 2];
7 }
8 void primeSieve() {
9     for (ll i = 3; i < N; i += 2) { // i * i < N reicht für isPrime
10         if (!isNotPrime[i / 2]) {
11             primes.push_back(i); // optional
12             for (ll j = i * i; j < N; j += 2 * i) {
13                 isNotPrime[j / 2] = 1;
14             }
15         }
16     }
17 }

```

#### 4.20 Möbius-Inversion

- Seien  $f, g: \mathbb{N} \rightarrow \mathbb{N}$  und  $g(n) := \sum_{d|n} f(d)$ . Dann ist  $f(n) = \sum_{d|n} g(d) \mu(\frac{n}{d})$ .

- $\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{falls } n=1 \\ 0 & \text{sonst} \end{cases}$

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  $\sum_{i=1}^N \mu(i) \cdot (2^{\text{cnt}[i]} - 1)$ .

#### 4.21 LGS über $\mathbb{F}_p$

gauss löst LGS  $O(n^3)$

```

1 void normalLine(int line, ll p) {
2     ll factor = multInv(mat[line][line], p);
3     for (ll& x : mat[line]) x = (x * factor) % p;
4 }
5 void takeAll(int n, int line, ll p) {
6     for (int i = 0; i < n; i++) {
7         if (i == line) continue;
8         ll diff = mat[i][line];
9         for (int j = 0; j < sz(mat[i]); j++) {
10             mat[i][j] -= (diff * mat[line][j]) % p;
11             mat[i][j] = (mat[i][j] + p) % p;
12         }
13 }
14 void gauss(int n, ll mod) {

```

```

14 vector<bool> done(n, false);
15 for (int i = 0; i < n; i++) {
16     int j = 0;
17     while (j < n && (done[j] || mat[j][i] == 0)) j++;
18     if (j == n) continue;
19     swap(mat[i], mat[j]);
20     normalLine(i, mod);
21     takeAll(n, i, mod);
22     done[i] = true;
23 }
24 // für Eindeutigkeit, Existenz etc. siehe LGS über R Seite 17

```

#### 4.22 LGS über $\mathbb{R}$

gauss löst LGS  $O(n^3)$

```

1 void normalLine(int line) {
2     double factor = mat[line][line];
3     for (double& x : mat[line]) x /= factor;
4 }
5 void takeAll(int n, int line) {
6     for (int i = 0; i < n; i++) {
7         if (i == line) continue;
8         double diff = mat[i][line];
9         for (int j = 0; j < sz(mat[i]); j++) {
10             mat[i][j] -= diff * mat[line][j];
11         }
12 }
13 int gauss(int n) {
14     vector<bool> done(n, false);
15     for (int i = 0; i < n; i++) {
16         int swappee = i; // Sucht Pivotzeile für bessere Stabilität.
17         for (int j = 0; j < n; j++) {
18             if (done[j]) continue;
19             if (abs(mat[j][i]) > abs(mat[i][i])) swappee = j;
20         }
21         swap(mat[i], mat[swappee]);
22         if (abs(mat[i][i]) > EPS) {
23             normalLine(i);
24             takeAll(n, i);
25             done[i] = true;
26         }
27         // Ab jetzt nur checks bzgl. Eindeutigkeit/Existenz der Lösung.
28         for (int i = 0; i < n; i++) {
29             bool allZero = true;
30             for (int j = i; j < n; j++) allZero &= abs(mat[i][j]) <= EPS;
31             if (allZero && abs(mat[i][n]) > EPS) return INCONSISTENT;
32             if (allZero && abs(mat[i][n]) <= EPS) return MULTIPLE;
33         }
34     }
35     return UNIQUE;
36 }

```

### 4.23 Numerisch Extremstelle bestimmen

```

1 template<typename F>
2 ld gss(ld l, ld r, F&& f) {
3     ld inv = (sqrt(5.0l) - 1) / 2;
4     ld x1 = r - inv*(r-l), x2 = l + inv*(r-l);
5     ld f1 = f(x1), f2 = f(x2);
6     for (int i = 0; i < 200; i++) {
7         if (f1 < f2) { //change to > to find maximum
8             r = x2; x2 = x1; f2 = f1;
9             x1 = r - inv*(r-l); f1 = f(x1);
10        } else {
11            l = x1; x1 = x2; f1 = f2;
12            x2 = l + inv*(r-l); f2 = f(x2);
13        }
14    }
15    return l;
16 }

```

### 4.24 Numerisch Integrieren, Simpsonregel

```

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

```

### 4.25 Polynome, FFT, NTT & andere Transformationen

Multipliziert Polynome  $A$  und  $B$ .

- $\deg(A \cdot B) = \deg(A) + \deg(B)$
- Vektoren  $a$  und  $b$  müssen mindestens Größe  $\deg(A \cdot B) + 1$  haben. Größe muss eine Zweierpotenz sein.
- Für ganzzahlige Koeffizienten: `(ll) round(real(a[i]))`
- xor*, *or* und *and* Transform funktioniert auch mit `double` oder modulo einer Primzahl  $p$  falls  $p \geq 2^{\text{bits}}$

```

1 using cplx = complex<double>;
2 void fft(vector<cplx>& a, bool inv = false) {
3     int n = sz(a);
4     for (int i = 0, j = 1; j < n - 1; ++j) {
5         for (int k = n >> 1; k > (i ^= k); k >>= 1);
6         if (j < i) swap(a[i], a[j]);
7     }
8     static vector<cplx> ws(2, 1);
9     for (static int k = 2; k < n; k *= 2) {
10        ws.resize(n);
11        cplx w = polar(1.0, acos(-1.0) / k);
12        for (int i = k; i < 2*k; i++) ws[i] = ws[i/2] * (i % 2 ? w : 1);
13    }
14    for (int s = 1; s < n; s *= 2) {
15        for (int j = 0; j < n; j += 2 * s) {
16            for (int k = 0; k < s; k++) {
17                cplx u = a[j + k], t = a[j + s + k];
18                t *= (inv ? conj(ws[s + k]) : ws[s + k]);

```

```

19        a[j + k] = u + t;
20        a[j + s + k] = u - t;
21        if (inv) a[j + k] /= 2, a[j + s + k] /= 2;
22    }
}

```

```

1 constexpr ll mod = 998244353, root = 3;
2 void ntt(vector<ll>& a, bool inv = false) {
3     int n = sz(a);
4     auto b = a;
5     ll r = inv ? powMod(root, mod - 2, mod) : root;
6     for (int s = n / 2; s > 0; s /= 2) {
7         ll ws = powMod(r, (mod - 1) / (n / s), mod), w = 1;
8         for (int j = 0; j < n / 2; j += s) {
9             for (int k = j; k < j + s; k++) {
10                ll u = a[j + k], t = a[j + s + k] * w % mod;
11                b[k] = (u + t) % mod;
12                b[n/2 + k] = (u - t + mod) % mod;
13            }
14            w = w * ws % mod;
15        }
16        swap(a, b);
17    }
18    if (inv) {
19        ll div = powMod(n, mod - 2, mod);
20        for (auto& x : a) x = x * div % mod;
21    }
}

```

```

1 void bitwiseConv(vector<ll>& a, bool inv = false) {
2     int n = sz(a);
3     for (int s = 1; s < n; s *= 2) {
4         for (int i = 0; i < n; i += 2 * s) {
5             for (int j = i; j < i + s; j++) {
6                 ll& u = a[j], &v = a[j + s];
7                 tie(u, v) = inv ? pair(v - u, u) : pair(v, u + v); // AND
8                 //tie(u, v) = inv ? pair(v, u - v) : pair(u + v, u); //OR
9                 //tie(u, v) = pair(u + v, u - v); // XOR
10            }
11        }
12        //if (inv) for (ll& x : a) x /= n; // XOR (careful with MOD)
}

```

Multiplikation mit 2 transforms statt 3: (nur benutzen wenn nötig!)

```

1 vector<cplx> mul(vector<ll>& a, vector<ll>& b) {
2     int n = 1 << (lg(sz(a) + sz(b) - 1) + 1);
3     vector<cplx> c(all(a)), d(n);
4     c.resize(n);
5     for (int i = 0; i < sz(b); i++) c[i] = {real(c[i]), b[i]};
6     fft(c);
7     for (int i = 0; i < n; i++) {
8         int j = (n - i) & (n - 1);
9         cplx x = (c[i] + conj(c[j])) / cplx{2, 0}; //fft(a)[i];
10        cplx y = (c[i] - conj(c[j])) / cplx{0, 2}; //fft(b)[i];
11        d[i] = x * y;
12    }
13    fft(d, true);
14    return d;
15 }

```

### 4.26 Operations on Formal Power Series

```

1 vector<ll> poly_inv(const vector<ll>& a, int n) {
2     vector<ll> q = {powMod(a[0], mod-2, mod)};
3     for (int len = 1; len < n; len *= 2) {
4         vector<ll> a2 = a, q2 = q;
5         a2.resize(2*len), q2.resize(2*len);
6         ntt(q2);
7         for (int j : {0, 1}) {
8             ntt(a2);
9             for (int i = 0; i < 2*len; i++) a2[i] = a2[i]*q2[i] % mod;
10            ntt(a2, true);
11            for (int i = 0; i < len; i++) a2[i] = 0;
12        }
13        for (int i = len; i < min(n, 2*len); i++) {
14            q.push_back((mod - a2[i]) % mod);
15        }
16        return q;
17    }
18 }
19 vector<ll> poly_deriv(vector<ll> a) {
20     for (int i = 1; i < sz(a); i++)
21         a[i-1] = a[i] * i % mod;
22     a.pop_back();
23     return a;
24 }
25 vector<ll> poly_integr(vector<ll> a) {
26     if (a.empty()) return {0};
27     a.push_back(a.back() * powMod(sz(a), mod-2, mod) % mod);
28     for (int i = sz(a)-2; i > 0; i--)
29         a[i] = a[i+1] * powMod(i, mod-2, mod) % mod;
30     a[0] = 0;
31     return a;
32 }
33 vector<ll> poly_log(vector<ll> a, int n) {
34     a = mul(poly_deriv(a), poly_inv(a, n));
35     a.resize(n-1);
36     a = poly_integr(a);
37     return a;
38 }
39 vector<ll> poly_exp(vector<ll> a, int n) {
40     vector<ll> q = {1};
41     for (int len = 1; len < n; len *= 2) {
42         vector<ll> p = poly_log(q, 2*len);
43         for (int i = 0; i < 2*len; i++)
44             p[i] = (mod - p[i] + (i < sz(a) ? a[i] : 0)) % mod;
45         vector<ll> q2 = q;
46         q2.resize(2*len);
47         ntt(p), ntt(q2);
48         for (int i = 0; i < 2*len; i++) p[i] = p[i] * q2[i] % mod;
49         ntt(p, true);
50         for (int i = len; i < min(n, 2*len); i++) q.push_back(p[i]);
51     }
52     return q;
}

```



## 4.27 Inversionszahl

```

1 ll inversions(const vector<ll>& v) {
2   Tree<pair<ll, ll>> t; //ordered statistics tree Seite 4
3   ll res = 0;
4   for (ll i = 0; i < sz(v); i++) {
5     res += i - t.order_of_key({v[i], i});
6     t.insert({v[i], i});
7   }
8   return res;
9 }

```

## 4.28 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  $g(X_1) \oplus \dots \oplus g(X_k)$ .

## 4.29 Kombinatorik

**Wilsons Theorem** A number  $n$  is prime if and only if  $(n-1)! \equiv -1 \pmod n$ .

( $n$  is prime if and only if  $(n-1)! \cdot (n-m)! \equiv (-1)^m \pmod n$  for all  $m$  in  $\{1, \dots, n\}$ )

$$(n-1)! \equiv \begin{cases} -1 \pmod n, & \text{falls } n \in \mathbb{P} \\ 2 \pmod n, & \text{falls } n=4 \\ 0 \pmod n, & \text{sonst} \end{cases}$$

**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.

**LUCAS-Theorem** Ist  $p$  prim,  $m = \sum_{i=0}^k m_i p^i$ ,  $n = \sum_{i=0}^k n_i p^i$  ( $p$ -adische Darstellung), so gilt

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{n_i} \pmod p.$$

**Binomialkoeffizienten** Die Anzahl der  $k$ -elementigen Teilmengen einer  $n$ -elementigen Menge.

precalc berechnet  $n!$  und  $n!^{-1}$  vor  $O(\text{lim})$

calc\_binom berechnet Binomialkoeffizient  $O(1)$

```

1 constexpr ll lim = 10'000'000;
2 ll fac[lim], inv[lim];
3 void precalc() {
4   fac[0] = inv[0] = 1;
5   for (int i = 1; i < lim; i++) fac[i] = fac[i-1] * i % mod;
6   inv[lim-1] = multInv(fac[lim-1], mod);
7   for (int i = lim-1; i > 0; i--) inv[i-1] = inv[i] * i % mod;
8 }
9 ll calc_binom(ll n, ll k) {
10  if (n < 0 || n < k || k < 0) return 0;
11  return (inv[k] * inv[n-k] % mod) * fac[n] % mod;
12 }

```

Falls  $n \geq p$  for  $\text{mod} = p^k$  berechne  $\text{fac}$  und  $\text{inv}$  aber teile  $p$  aus  $i$  und berechne die Häufigkeit von  $p$  in  $n!$  als  $\sum_{i=1}^n \left\lfloor \frac{n}{p^i} \right\rfloor$

calc\_binom berechnet Binomialkoeffizient ( $n \leq 61$ )  $O(k)$

```

1 ll calc_binom(ll n, ll k) {
2   if (k > n) return 0;
3   ll r = 1;
4   for (ll d = 1; d <= k; d++) { // Reihenfolge => Teilbarkeit
5     r *= n--, r /= d;
6   }
7   return r;
8 }

```

calc\_binom berechnet Binomialkoeffizient modulo Primzahl  $p$   $O(p-n)$

```

1 ll calc_binom(ll n, ll k, ll p) {
2   assert(n < p); //wichtig: sonst falsch!
3   if (k > n) return 0;
4   ll x = k % 2 != 0 ? p-1 : 1;
5   for (ll c = p-1; c > n; c--) {
6     x *= c - k; x %= p;
7     x *= multInv(c, p); x %= p;
8   }
9   return x;
10 }

```

### CATALAN-Zahlen

• Die CATALAN-Zahl  $C_n$  gibt an:

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

$$C_0 = 1 \quad C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1} \binom{2n}{n} = \frac{4n-2}{n+1} \cdot C_{n-1}$$

- Formel 1 erlaubt Berechnung ohne Division in  $O(n^2)$
- Formel 2 und 3 erlauben Berechnung in  $O(n)$

### CATALAN-Convolution

• Anzahl an Klammerausdrücken mit  $n+k$  Klammerpaaren, die mit  $\binom{k}{\cdot}$  beginnen.

$$C_0^k = 1 \quad C_n^k = \sum_{a_0+a_1+\dots+a_k=n} C_{a_0} C_{a_1} \dots C_{a_k} = \frac{k+1}{n+k+1} \binom{2n+k}{n} = \frac{(2n+k-1) \cdot (2n+k)}{n(n+k+1)} \cdot C_{n-1}$$

**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.

$$\langle n \rangle = \langle n-1 \rangle = 1 \quad \langle n \rangle_k = (k+1) \langle n-1 \rangle_k + (n-k) \langle n-1 \rangle_{k-1} = \sum_{i=0}^k (-1)^i \binom{n+1}{i} (k+1-i)^n$$

- Formel 1 erlaubt Berechnung ohne Division in  $O(n^2)$
- Formel 2 erlaubt Berechnung in  $O(n \log(n))$

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

$$\langle\langle n \rangle\rangle = 1 \quad \langle\langle n \rangle\rangle_k = 0 \quad \langle\langle n \rangle\rangle_k = (k+1) \langle\langle n-1 \rangle\rangle_k + (2n-k-1) \langle\langle n-1 \rangle\rangle_{k-1}$$

- Formel erlaubt Berechnung ohne Division in  $O(n^2)$

**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.

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1 \quad \begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0 \quad \begin{bmatrix} n \\ k \end{bmatrix} = \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} + (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix}$$

- Formel erlaubt Berechnung ohne Division in  $O(n^2)$

$$\sum_{k=0}^n \pm \begin{bmatrix} n \\ k \end{bmatrix} x^k = x(x-1)(x-2) \dots (x-n+1)$$

- Berechne Polynom mit FFT und benutze Betrag der Koeffizienten  $O(n \log(n)^2)$  (nur ungefähr gleich große Polynome zusammen multiplizieren beginnend mit  $x-k$ )

**STIRLING-Zahlen 2. Ordnung** Die Anzahl der Möglichkeiten  $n$  Elemente in  $k$  nicht-leere Teilmengen zu zerlegen. Es gibt  $k$  Möglichkeiten die  $n$  in eine  $n-1$ -Partition einzuordnen. Dazu kommt der Fall, dass die  $n$  in ihrer eigenen Teilmenge (alleine) steht.

$$\begin{Bmatrix} n \\ 1 \end{Bmatrix} = \begin{Bmatrix} n \\ n \end{Bmatrix} = 1 \quad \begin{Bmatrix} n \\ k \end{Bmatrix} = k \begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix} = \frac{1}{k!} \sum_{i=0}^k (-1)^{k-i} \binom{k}{i} i^n$$

- Formel 1 erlaubt Berechnung ohne Division in  $O(n^2)$
- Formel 2 erlaubt Berechnung in  $O(n \log(n))$

**BELL-Zahlen** Anzahl der Partitionen von  $\{1, \dots, n\}$ . Wie STIRLING-Zahlen 2. Ordnung ohne Limit durch  $k$ .

$$B_1 = 1 \quad B_n = \sum_{k=0}^{n-1} B_k \begin{Bmatrix} n-1 \\ k \end{Bmatrix} = \sum_{k=0}^n \begin{Bmatrix} n \\ k \end{Bmatrix} \quad B_{p^m+n} \equiv m \cdot B_n + B_{n+1} \pmod p$$

**Partitions** Die Anzahl der Partitionen von  $n$  in genau  $k$  positive Summanden. Die Anzahl der Partitionen von  $n$  mit Elementen aus  $1, \dots, k$ .

$$p_0(0) = 1 \quad p_k(n) = 0 \text{ für } k > n \text{ oder } n \leq 0 \text{ oder } k \leq 0$$

$$p_k(n) = p_k(n-k) + p_{k-1}(n-1)$$

$$p(n) = \sum_{k=1}^n p_k(n) = p_n(2n) = \sum_{k \neq 0}^{\infty} (-1)^{k+1} p\left(n - \frac{k(3k-1)}{2}\right)$$

- in Formel 3 kann abgebrochen werden wenn  $\frac{k(3k-1)}{2} > n$ .
- Die Anzahl der Partitionen von  $n$  in bis zu  $k$  positive Summanden ist  $\sum_{i=0}^k p_i(n) = p_k(n+k)$ .

### 4.30 The Twelffold Way (verteile $n$ Bälle auf $k$ Boxen)

Bälle Boxen	identisch identisch	verschieden identisch	identisch verschieden	verschieden verschieden
–	$p_k(n+k)$	$\sum_{i=0}^k \begin{Bmatrix} n \\ i \end{Bmatrix}$	$\begin{Bmatrix} n+k-1 \\ k-1 \end{Bmatrix}$	$k^n$
Bälle pro Box $\geq 1$	$p_k(n)$	$\begin{Bmatrix} n \\ k \end{Bmatrix}$	$\begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}$	$k! \begin{Bmatrix} n \\ k \end{Bmatrix}$
Bälle pro Box $\leq 1$	$[n \leq k]$	$[n \leq k]$	$\binom{k}{n}$	$n! \binom{k}{n}$
[Bedingung]: <b>return</b> Bedingung ? 1 : 0;				

Binomialkoeffizienten			
$\frac{n!}{k!(n-k)!} = \binom{n}{k} = \binom{n}{n-k} = \frac{n}{k} \binom{n-1}{k-1} = \frac{n-k+1}{k} \binom{n}{k-1} = \binom{n-1}{k} + \binom{n-1}{k-1} = (-1)^k \binom{k-n-1}{k} \approx 2^n \cdot \frac{-2}{\sqrt{2\pi n}} \cdot \exp\left(-\frac{2(x-\frac{n}{2})^2}{n}\right)$			
$\sum_{k=0}^n \binom{n}{k} = 2^n$	$\sum_{k=0}^n \binom{k}{m} = \binom{n+1}{m+1}$	$\sum_{i=0}^n \binom{n}{i}^2 = \binom{2n}{n}$	$\sum_{k=0}^n \binom{r+k}{k} = \binom{r+n+1}{n}$
$\binom{n}{m} \binom{m}{k} = \binom{n}{k} \binom{n-k}{m-k}$	$\sum_{k=0}^n \binom{r}{k} \binom{s}{n-k} = \binom{r+s}{n}$	$\sum_{i=1}^n \binom{n}{i} F_i = F_{2n}$	$F_n = n\text{-th Fib.}$

Important Numbers						
10 <sup>x</sup>	Highly Composite	# Divs	< Prime	> Prime	# Primes	primorial
1	6	4	-3	+1	4	2
2	60	12	-3	+1	25	3
3	840	32	-3	+9	168	4
4	7560	64	-27	+7	1229	5
5	83160	128	-9	+3	9592	6
6	720720	240	-17	+3	78498	7
7	8648640	448	-9	+19	664579	8
8	73513440	768	-11	+7	5761455	8
9	735134400	1344	-63	+7	50847534	9
10	6983776800	2304	-33	+19	455052511	10
11	97772875200	4032	-23	+3	4118054813	10
12	96376198400	6720	-11	+39	37607912018	11
13	9316358251200	10752	-29	+37	346065536839	12
14	97821761637600	17280	-27	+31	3204941750802	12
15	866421317361600	26880	-11	+37	29844570422669	13
16	8086598962041600	41472	-63	+61	279238341033925	13
17	74801040398884800	64512	-3	+3	2623557157654233	14
18	897612484786617600	103680	-11	+3	24739954287740860	16

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)	$(n^{30}+15n^{16}+20n^{10}+24n^6)/60$			

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

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

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

Verschiedenes	
Türme von Hanoi, minimale Schrittzahl:	$T_n = 2^n - 1$
#Regionen zwischen $n$ Geraden	$\frac{n(n+1)}{2} + 1$
#abgeschlossene Regionen zwischen $n$ Geraden	$\frac{n^2-3n+2}{2}$
#markierte, gewurzelte Bäume	$n^{n-1}$
#markierte, nicht gewurzelte Bäume	$n^{n-2}$
#Wälder mit $k$ gewurzelten Bäumen	$\frac{k}{n} \binom{n}{k} n^{n-k}$
#Wälder mit $k$ gewurzelten Bäumen mit vorgegebenen Wurzelknoten	$\frac{k}{n} n^{n-k}$
Derangements	$!n = (n-1)!(n-1) + (n-2)! = \left\lfloor \frac{n!}{e} + \frac{1}{2} \right\rfloor$
	$\lim_{n \rightarrow \infty} \frac{!n}{n!} = \frac{1}{e}$

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 $SG_0 = 0, SG_n = \lfloor \log_2 n \rfloor + 1$
$M_{\text{❶}} = \text{Teiler von } pile_i$ $M_{\text{❷}} = \text{echte Teiler von } pile_i$	❶ $SG_0 = 0, SG_n = SG_{\text{❷}, n} + 1$ 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. $n \in [72, 83]: \quad 4, 1, 2, 8, 1, 4, 7, 2, 1, 8, 2, 7$ Periode ab $n = 72$ der Länge 12.

## 5 Strings

### 5.1 KNUTH-MORRIS-PRATT-Algorithmus

kmpSearch sucht sub in s  $O(|s|+|sub|)$

```
1 vector<int> kmpPreprocessing(const string& sub) {
2     vector<int> b(sz(sub) + 1);
3     b[0] = -1;
4     for (int i = 0, j = -1; i < sz(sub);) {
5         while (j >= 0 && sub[i] != sub[j]) j = b[j];
6         b[++i] = ++j;
7     }
8     return b;
9 }
10 vector<int> kmpSearch(const string& s, const string& sub) {
11     vector<int> result, pre = kmpPreprocessing(sub);
12     for (int i = 0, j = 0; i < sz(s);) {
13         while (j >= 0 && s[i] != sub[j]) j = pre[j];
14         i++; j++;
15         if (j == sz(sub)) {
16             result.push_back(i - j);
17             j = pre[j];
18         }
19     }
20     return result;
21 }
```

### 5.2 Z-Algorithmus

$z_i$  := Längstes gemeinsames Präfix von  $s_0 \dots s_{i-1}$  und  $s_i \dots s_{n-1}$   $O(n)$

Suchen: Z-Algorithmus auf P\$S ausführen, Positionen mit  $z_i = |P|$  zurückgeben

```
1 vector<int> Z(const string& s) {
2     int n = sz(s);
3     vector<int> z(n);
4     for (int i = 1, x = 0; i < n; i++) {
5         z[i] = max(0, min(z[i - x], x + z[x] - i));
6         while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
7             x = i, z[i]++;
8         }
9     }
10    return z;
11 }
```

### 5.3 Rolling Hash

```
1 // M = 1.7e9 + 9, 1e18L + 9, 2.2e18L + 7
2 struct Hash {
3     static constexpr ll M = 3e18L + 37;
4     static constexpr ll Q = 318LL << 53; // Random in [SIGMA+1, M]
5     vector<ll> pref = {0}, power = {1};
6     Hash(const string& s) {
7         for (auto c : s) { // c > 0
8             pref.push_back((mul(pref.back(), Q) + c + M) % M);
9             power.push_back(mul(power.back(), Q));
10        }
11        ll operator()(int l, int r) {
12            return (pref[r] - mul(power[r-l], pref[l]) + M) % M;
13        }
14        static ll mul(__int128 a, ll b) {return a * b % M;}
15    };
```

### 5.4 Pattern Matching mit Wildcards

Gegeben zwei strings A und B, B enthält k wildcards enthält. Sei:

$$a_i = \cos(\alpha_i) + i \sin(\alpha_i) \quad \text{mit } \alpha_i = \frac{2\pi A[i]}{\Sigma}$$

$$b_i = \cos(\beta_i) + i \sin(\beta_i) \quad \text{mit } \beta_i = \begin{cases} \frac{2\pi B[|B|-i-1]}{\Sigma} & \text{falls } B[|B|-i-1] \in \Sigma \\ 0 & \text{sonst} \end{cases}$$

B matcht A an stelle i wenn  $(b \cdot a)[|B|-1+i] = |B|-k$ . Benutze FFT um  $(b \cdot a)$  zu berechnen.

### 5.5 MANACHER's Algorithm, Longest Palindrome

init transformiert string a  $O(n)$   
manacher berechnet Längen der Palindrome in longest  $O(n)$

```
1 vector<int> manacher(const string& t) {
2     //transforms "aa" to ".a.a." to find even length palindromes
3     string s(sz(t) * 2 + 1, '.');
4     for (int i = 0; i < sz(t); i++) s[2 * i + 1] = t[i];
5     int mid = 0, r = 0, n = sz(s);
6     vector<int> pal(n);
7     for (int i = 1; i < n - 1; i++) {
8         if (r > i) pal[i] = min(r - i, pal[2 * mid - i]);
9         while (pal[i] < min(i, n - i - 1) &&
10              s[i + pal[i] + 1] == s[i - pal[i] - 1]) {
11             pal[i]++;
12         }
13         if (i + pal[i] > r) mid = i, r = i + pal[i];
14     }
15     //convert lengths to constructed string s (optional)
16     //for (int i = 0; i < n; i++) pal[i] = 2 * pal[i] + 1;
17     return pal;
18 }
```

### 5.6 Longest Common Subsequence

lcscs findet längste gemeinsame Sequenz  $O(|a| \cdot |b|)$

```
1 string lcscs(const string& a, const string& b) {
2     vector<vector<int>> m(sz(a) + 1, vector<int>(sz(b) + 1));
3     for (int i = sz(a) - 1; i >= 0; i--) {
4         for (int j = sz(b) - 1; j >= 0; j--) {
5             if (a[i] == b[j]) m[i][j] = 1 + m[i+1][j+1];
6             else m[i][j] = max(m[i+1][j], m[i][j+1]);
7         } // Für die Länge: return m[0][0];
8     }
9     string res;
10    for (int j = 0, i = 0; j < sz(b) && i < sz(a);) {
11        if (a[i] == b[j]) res += a[i++], j++;
12        else if (m[i][j+1] > m[i+1][j]) j++;
13        else i++;
14    }
15    return res;
16 }
```

### 5.7 AHO-CORASICK-Automat

sucht patterns im Text  $O(|Text| + \sum |pattern|)$

1. mit addString(pattern, idx) Patterns hinzufügen.
2. rufe buildGraph() auf
3. mit state = go(state, idx) in nächsten Zustand wechseln.
4. erhöhe dabei dp[state]++
5. rufe dfs() auf. In dp[pattern state] stehen die Anzahl der Matches

```
1 constexpr ll ALPHABET_SIZE = 26, OFFSET = 'a';
2 struct AhoCorasick {
3     struct vert {
4         int suffix = 0, ch, cnt = 0;
5         array<int, ALPHABET_SIZE> nxt = {};
6         vert(int p, int c) : suffix(-p), ch(c) {}
7     };
8     vector<vert> aho = {{0, -1}};
9     int addString(string &s) {
10        int v = 0;
11        for (auto c : s) {
12            int idx = c - OFFSET;
13            if (!aho[v].nxt[idx]) {
14                aho[v].nxt[idx] = sz(aho);
15                aho.emplace_back(v, idx);
16            }
17            v = aho[v].nxt[idx];
18        }
19        aho[v].cnt++;
20        return v; // trie node index of pattern (pattern state)
21    }
22    int getSuffix(int v) {
23        if (aho[v].suffix < 0) {
24            aho[v].suffix = go(getSuffix(-aho[v].suffix), aho[v].ch);
25        }
26        return aho[v].suffix;
27    }
28    int go(int v, int idx) { // Root is v=0, idx is char - OFFSET
29        if (aho[v].nxt[idx]) return aho[v].nxt[idx];
30        else return v == 0 ? 0 : go(getSuffix(v), idx);
31    }
32    vector<vector<int>> adj;
33    vector<ll> dp;
34    void buildGraph() {
35        adj.resize(sz(aho));
36        dp.assign(sz(aho), 0);
37        for (int i = 1; i < sz(aho); i++) {
38            adj[getSuffix(i)].push_back(i);
39        }
40        void dfs(int v = 0) { // dp on tree
41            for (int u : adj[v]) {
42                //dp[u] = dp[v] + aho[u].cnt; // pattern count
43                dfs(u);
44                dp[v] += dp[u]; // no of matches
45            }
46        };
47    }
```

## 5.8 Lyndon und De-Bruijn

- **Lyndon-Wort:** Ein Wort das lexikographisch kleiner ist als jede seiner Rotationen.
  - Jedes Wort kann *eindeutig* in eine nicht ansteigende Folge von Lyndon-Worten zerlegt werden.
  - Für Lyndon-Worte  $u, v$  mit  $u < v$  gilt, dass  $uv$  auch ein Lyndon-Wort ist.
- next      lexikographisch nächstes Lyndon-Wort     $O(n)$ , Durchschnitt  $\Theta(1)$   
 duval      zerlegt  $s$  in Lyndon-Worte     $O(n)$   
 minrotation    berechnet kleinste Rotation von  $s$      $O(n)$

```

1 bool next(string& s, int maxLen, char mi = '0', char ma = '1') {
2     for (int i = sz(s), j = sz(s); i < maxLen; i++)
3         s.push_back(s[i % j]);
4     while(!s.empty() && s.back() == ma) s.pop_back();
5     if (s.empty()) {
6         s = mi;
7         return false;
8     } else {
9         s.back()++;
10        return true;
11    }
}

```

```

1 vector<pair<int, int>> duval(const string& s) {
2     vector<pair<int, int>> res;
3     for (int i = 0; i < sz(s);) {
4         int j = i + 1, k = i;
5         for (; j < sz(s) && s[k] <= s[j]; j++) {
6             if (s[k] < s[j]) k = j;
7             else k++;
8         }
9         while (i <= k) {
10            res.push_back({i, i + j - k});
11            i += j - k;
12        }
13        return res;
14    }
}

```

```

15 int minrotation(const string& s) {
16     auto parts = duval(s+s);
17     for (auto [l, r] : parts) {
18         if (l < sz(s) && r >= sz(s)) {
19             return l;
20         }
21     }
}

```

- **De-Bruijn-Sequenz  $B(\Sigma, n)$ :** ein Wort das jedes Wort der Länge  $n$  genau einmal als substring enthält (und minimal ist). Wobei  $B(\Sigma, n)$  zyklisch betrachtet wird.
  - es gibt  $\frac{(k!)^{n-1}}{k^n}$  verschiedene  $B(\Sigma, n)$
  - $B(\Sigma, n)$  hat Länge  $|\Sigma|^n$
- deBruijn berechnet ein festes  $B(\Sigma, n)$   $O(|\Sigma|^n)$

```

1 string deBruijn(int n, char mi = '0', char ma = '1') {
2     string res, c(1, mi);
3     do {
4         if (n % sz(c) == 0) res += c;
5     } while(next(c, n, mi, ma));
6     return res;
7 }

```

## 5.9 Suffix-Array

SuffixArray    berechnet ein Suffix Array     $O(|s| \cdot \log^2(|s|))$   
 lcp      berechnet Länge des longest common prefix     $O(\log(|s|))$   
             von  $s[x]$  und  $s[y]$

```

1 constexpr int MAX_CHAR = 256;
2 struct SuffixArray {
3     int n;
4     vector<int> SA, LCP;
5     vector<vector<int>> P;
6
7     SuffixArray(const string& s) : n(sz(s)), SA(n), LCP(n),
8         P(_lg(2 * n - 1) + 1, vector<int>(n)) {
9         P[0].assign(all(s));
10        iota(all(SA), 0);
11        sort(all(SA), [&](int a, int b) {return s[a] < s[b];});
12        vector<int> x(n);
13        for (int k = 1, c = 1; c < n; k++, c *= 2) {
14            iota(all(x), n - c);
15            for (int ptr = c; int i : SA) if (i >= c) x[ptr++] = i - c;
16            vector<int> cnt(k == 1 ? MAX_CHAR : n);
17            for (int i : P[k-1]) cnt[i]++;
18            partial_sum(all(cnt), begin(cnt));
19            for (int i : x | views::reverse) SA[--cnt[P[k-1][i]]] = i;
20            auto p = [&](int i) {return i < n ? P[k-1][i] : -1;};
21            for (int i = 1; i < n; i++) {
22                int a = SA[i-1], b = SA[i];
23                P[k][b] = P[k][a] + (p(a) != p(b) || p(a+c) != p(b+c));
24            }
25            for (int i = 1; i < n; i++) LCP[i] = lcp(SA[i-1], SA[i]);
26        }
27        int lcp(int x, int y) { // x & y are text-indices, not SA-indices
28            if (x == y) return n - x;
29            int res = 0;
30            for (int i = sz(P) - 1; i >= 0 && max(x, y) + res < n; i--) {
31                if (P[i][x + res] == P[i][y + res]) res |= 1 << i;
32            }
33            return res;
34        }
35    };
}

```

## 5.10 Suffix-Baum

SuffixTree    berechnet einen Suffixbaum     $O(|s|)$   
 extend      fügt den nächsten Buchstaben aus  $s$  ein     $O(1)$

```

1 struct SuffixTree {
2     struct Vert {
3         int start, end, suf; // s[start...end] along parent edge
4         map<char, int> nxt;
5     };
6     string s;
7     int needsSuffix, pos, remainder, curVert, curEdge, curLen;
8     // Each Vertex gives its children range as [start, end]
9     vector<Vert> tree = {Vert{-1, -1, 0, {}}};
10
11     SuffixTree(const string& s_) : s(s_) {
12         needsSuffix = remainder = curVert = curEdge = curLen = 0;
13         pos = -1;
14         for (int i = 0; i < sz(s); i++) extend();
15     }
}

```

```

14 }
15 int newVert(int start, int end) {
16     tree.push_back({start, end, 0, {}});
17     return sz(tree) - 1;
18 }
19 void addSuffixLink(int vert) {
20     if (needsSuffix) tree[needsSuffix].suf = vert;
21     needsSuffix = vert;
22 }
23 bool fullImplicitEdge(int vert) {
24     int len = min(tree[vert].end, pos + 1) - tree[vert].start;
25     if (curLen >= len) {
26         curEdge += len;
27         curLen -= len;
28         curVert = vert;
29         return true;
30     } else {
31         return false;
32     }
33 }
34 void extend() {
35     pos++;
36     needsSuffix = 0;
37     remainder++;
38     while (remainder) {
39         if (curLen == 0) curEdge = pos;
40         if (!tree[curVert].nxt.count(s[curEdge])) {
41             int leaf = newVert(pos, sz(s));
42             tree[curVert].nxt[s[curEdge]] = leaf;
43             addSuffixLink(curVert);
44         } else {
45             int nxt = tree[curVert].nxt[s[curEdge]];
46             if (fullImplicitEdge(nxt)) continue;
47             if (s[tree[nxt].start + curLen] == s[pos]) {
48                 curLen++;
49                 addSuffixLink(curVert);
50                 break;
51             }
52             int split = newVert(tree[nxt].start,
53                 tree[nxt].start + curLen);
54             tree[curVert].nxt[s[curEdge]] = split;
55             int leaf = newVert(pos, sz(s));
56             tree[split].nxt[s[pos]] = leaf;
57             tree[nxt].start += curLen;
58             tree[split].nxt[s[tree[nxt].start]] = nxt;
59             addSuffixLink(split);
60         }
61         remainder--;
62         if (curVert == 0 && curLen) {
63             curLen--;
64             curEdge = pos - remainder + 1;
65         } else {
66             curVert = tree[curVert].suf ? tree[curVert].suf : 0;
67         }
68     }
}

```

## 5.11 Suffix-Automaton

- 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 und prüfe, ob Endzustand 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.

```

1 constexpr int ALPHABET_SIZE = 26;
2 constexpr char OFFSET = 'a';
3 struct SuffixAutomaton {
4     struct State {
5         int len, link = -1;
6         array<int, ALPHABET_SIZE> nxt; // map if large Alphabet
7         State(int l) : len(l) {fill(all(nxt), -1);}
8     };
9     vector<State> st = {State(0)};
10    int cur = 0;
11    SuffixAutomaton(const string& s) {
12        st.reserve(2 * sz(s));
13        for (auto c : s) extend(c - OFFSET);
14    }
15    void extend(int c) {
16        int p = cur;
17        cur = sz(st);
18        st.emplace_back(st[p].len + 1);
19        for (; p != -1 && st[p].nxt[c] < 0; p = st[p].link) {
20            st[p].nxt[c] = cur;
21        }
22        if (p == -1) {
23            st[cur].link = 0;
24        } else {
25            int q = st[p].nxt[c];
26            if (st[p].len + 1 == st[q].len) {
27                st[cur].link = q;
28            } else {
29                st.emplace_back(st[p].len + 1);
30                st.back().link = st[q].link;
31                st.back().nxt = st[q].nxt;
32                for (; p != -1 && st[p].nxt[c] == q; p = st[p].link) {
33                    st[p].nxt[c] = sz(st) - 1;
34                }
35                st[q].link = st[cur].link = sz(st) - 1;
36            }
37        }
38        vector<int> calculateTerminals() {
39            vector<int> terminals;
40            for (int p = cur; p != -1; p = st[p].link) {
41                terminals.push_back(p);
42            }
43            return terminals;
44        }
45    }

```

```

44 // Pair with start index (in t) and length of LCS.
45 pair<int, int> longestCommonSubstring(const string& t) {
46     int v = 0, l = 0, best = 0, bestp = -1;
47     for (int i = 0; i < sz(t); i++) {
48         int c = t[i] - OFFSET;
49         while (v > 0 && st[v].nxt[c] < 0) {
50             v = st[v].link;
51             l = st[v].len;
52         }
53         if (st[v].nxt[c] >= 0) v = st[v].nxt[c], l++;
54         if (l > best) best = l, bestp = i;
55     }
56     return {bestp - best + 1, best};
57 }
58 }

```

## 5.12 Trie

```

1 // Zahlenwerte müssen bei 0 beginnen und zusammenhängend sein.
2 constexpr int ALPHABET_SIZE = 2;
3 struct node {
4     int words, ends;
5     array<int, ALPHABET_SIZE> nxt;
6     node() : words(0), ends(0) {fill(all(nxt), -1);}
7 };
8 vector<node> trie = {node()};
9 int traverse(const vector<int>& word, int x) {
10    int id = 0;
11    for (int c : word) {
12        if (id < 0 || (trie[id].words == 0 && x <= 0)) return -1;
13        trie[id].words += x;
14        if (trie[id].nxt[c] < 0 && x > 0) {
15            trie[id].nxt[c] = sz(trie);
16            trie.emplace_back();
17        }
18        id = trie[id].nxt[c];
19    }
20    trie[id].words += x;
21    trie[id].ends += x;
22    return id;
23 }
24 int insert(const vector<int>& word) {
25     return traverse(word, 1);
26 }
27 bool erase(const vector<int>& word) {
28     int id = traverse(word, 0);
29     if (id < 0 || trie[id].ends <= 0) return false;
30     traverse(word, -1);
31     return true;
32 }

```

## 6 Python

### 6.1 Recursion

```

1 import sys
2 sys.setrecursionlimit(1000000)

```

## 6.2 IO

```

1 n, m = map(int, input().split())
2 A = list(map(int, input().split()))
3 print(n, m, *A)

```

## 7 Sonstiges

### 7.1 Compiletime

- überprüfen ob Compilezeit Berechnungen erlaubt sind!
- braucht c++14 oder höher!

```

1 template<int N>
2 struct Table {
3     int data[N];
4     constexpr Table() : data {} {
5         for (int i = 0; i < N; i++) data[i] = i;
6     };
7     constexpr Table<100'000> precalculated;

```

### 7.2 Timed

Kann benutzt werden um randomisierte Algorithmen so lange wie möglich laufen zu lassen.

```

1 int times = clock();
2 //run for 900ms
3 while (1000*(clock()-times)/CLOCKS_PER_SEC < 900) {...}

```

### 7.3 Bit Operations

Bit an Position j lesen	$(x \& (1 \ll j)) \neq 0$
Bit an Position j setzten	$x \mid= (1 \ll j)$
Bit an Position j löschen	$x \&= \sim(1 \ll j)$
Bit an Position j flippen	$x \hat{=} (1 \ll j)$
Anzahl an führenden nullen ( $x \neq 0$ )	<code>__builtin_clzll(x)</code>
Anzahl an schließenden nullen ( $x \neq 0$ )	<code>__builtin_ctzll(x)</code>
Anzahl an 1 bits	<code>__builtin_popcountll(x)</code>
i-te Zahl eines Graycodes	$i \wedge (i \gg 1)$

```

1 // Iteriert über alle Teilmengen einer Bitmaske
2 // (außer der leeren Menge).
3 for (int subset = bitmask; subset > 0;
4      subset = (subset - 1) & bitmask)
5 // Zählt Anzahl der gesetzten Bits.
6 int numberOfSetBits(int i) {
7     i = i - ((i >> 1) & 0x5555'5555);
8     i = (i & 0x3333'3333) + ((i >> 2) & 0x3333'3333);
9     return (((i + (i >> 4)) & 0x0F0F'0F0F) * 0x0101'0101) >> 24;
10 }
11 // Nächste Permutation in Bitmaske
12 // (z.B. 00111 => 01011 => 01101 => ...)
13 ll nextPerm(ll v) {
14     ll t = v | (v - 1);
15     return (t+1) | (((~t & --t) - 1) >> (__builtin_ctzll(v) + 1));
16 }

```



## 7.4 Overflow-sichere arithmetische Operationen

Gibt zurück, ob es einen Overflow gab. Wenn nicht, enthält `c` das Ergebnis.

Addition	<code>__builtin_saddll_overflow(a, b, &amp;c)</code>
Subtraktion	<code>__builtin_ssubll_overflow(a, b, &amp;c)</code>
Multiplikation	<code>__builtin_smulll_overflow(a, b, &amp;c)</code>

## 7.5 Pragmas

```
1 #pragma GCC optimize("Ofast")
2 #pragma GCC optimize ("unroll-loops")
3 #pragma GCC target("sse,sse2,sse3,ssse3,sse4,"
4 "popcnt,abm,mmx,avx,tune=native")
5 #pragma GCC target("fpmath=sse,sse2") // no excess precision
6 #pragma GCC target("fpmath=387") // force excess precision
```

## 7.6 DP Optimizations

Aufgabe: Partitioniere Array in genau  $m$  zusammenhängende Teile mit minimalen Kosten:  $dp[i][j] = \min_{k < j} \{dp[i-1][k-1] + C[k][j]\}$ . Es sei  $A[i][j]$  das *minimale* optimale  $k$  bei der Berechnung von  $dp[i][j]$ .

**Knuth-Optimization** Vorbedingung:  $A[i-1][j] \leq A[i][j] \leq A[i][j+1]$   
`calc` berechnet das DP  $O(n^2)$

```
1 ll calc(int n, int m, const vector<vector<ll>>& C) {
2     vector<vector<ll>> dp(m, vector<ll>(n, inf));
3     vector<vector<int>> opt(m, vector<int>(n + 1, n - 1));
4     for (int i = 0; i < n; i++) dp[0][i] = C[0][i];
5     for (int i = 1; i < m; i++) {
6         for (int j = n - 1; j >= 0; --j) {
7             opt[i][j] = i == 1 ? 0 : opt[i - 1][j];
8             for (int k = opt[i][j]; k <= min(opt[i][j+1], j-1); k++) {
9                 if (dp[i][j] <= dp[i - 1][k] + C[k + 1][j]) continue;
10                dp[i][j] = dp[i - 1][k] + C[k + 1][j];
11                opt[i][j] = k;
12            }
13            return dp[m - 1][n - 1];
14 }
```

**Divide and Conquer** Vorbedingung:  $A[i][j-1] \leq A[i][j]$ .  
`calc` berechnet das DP  $O(m \cdot n \cdot \log(n))$

```
1 vector<vector<ll>> dp;
2 vector<vector<ll>> C;
3 void rec(int i, int j0, int j1, int m0, int m1) {
4     if (j1 < j0) return;
5     int jmid = (j0 + j1) / 2;
6     dp[i][jmid] = inf;
7     int bestk = m0;
8     for (int k = m0; k < min(jmid, m1 + 1); ++k) {
9         if (dp[i - 1][k] + C[k + 1][jmid] < dp[i][jmid]) {
10            dp[i][jmid] = dp[i - 1][k] + C[k + 1][jmid];
11            bestk = k;
12        }
13        rec(i, j0, jmid - 1, m0, bestk);
14        rec(i, jmid + 1, j1, bestk, m1);
15    }
16    ll calc(int n, int m) {
17        dp = vector<vector<ll>>(m, vector<ll>(n, inf));
18        for (int i = 0; i < n; i++) dp[0][i] = C[0][i];
```

```
19     for (int i = 1; i < m; i++) {
20         rec(i, 0, n - 1, 0, n - 1);
21     }
22     return dp[m - 1][n - 1];
23 }
```

**Quadrangle inequality** Die Bedingung  $\forall a \leq b \leq c \leq d : C[a][d] + C[b][c] \geq C[a][c] + C[b][d]$  ist hinreichend für beide Optimierungen.

**Sum over Subsets DP**  $res[mask] = \sum_{i \subseteq mask} in[i]$ . Für Summe über Supersets `res` einmal vorher und einmal nachher reversen.

```
1 vector<ll> res(in);
2 for (int i = 1; i < sz(res); i *= 2) {
3     for (int mask = 0; mask < sz(res); mask++) {
4         if (mask & i) {
5             res[mask] += res[mask ^ i];
6         }
7     }
```

## 7.7 Parallel Binary Search

```
1 // Q = # of queries, bucket sort is sometimes faster
2 vector<int> low(Q, 0), high(Q, MAX_OPERATIONS);
3 while (true) {
4     vector<pair<int, int>> focus;
5     for (int i = 0; i < Q; i++) if (low[i] < high[i]) {
6         focus.emplace_back((low[i] + high[i]) / 2, i);
7     }
8     if (focus.empty()) break;
9     sort(all(focus));
10    // reset simulation
11    for (int step = 0; auto [mid, i] : focus) {
12        while (step <= mid) {
13            // simulation step
14            step++;
15        }
16        if (/* requirement already fulfilled */) high[i] = mid;
17        else low[i] = mid + 1;
18    } // answer in low (and high)
```

## 7.8 Josephus-Problem

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

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

```
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     }
8     return n <= 1; n++; return n;
```

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

```
1 // Der letzte Überlebende, 0-basiert.
2 int josephus(int n, int k) {
3     if (n == 1) return 0;
4     return (josephus(n - 1, k) + k) % n;
5 }
```

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

## 7.9 Sonstiges

```
1 // Alles-Header.
2 #include <bits/stdc++.h>
3 // Setzt deutsche Tastaturlayout / toggle mit alt + space
4 setxkbmap de
5 setxkbmap de,us -option grp:alt_space_toggle
6 // Schnelle Ein-/Ausgabe mit cin/cout.
7 cin.tie(nullptr)->ios::sync_with_stdio(false);
8 // Set mit eigener Sortierfunktion.
9 set<point2, decltype(comp)> set1(comp);
10 // STL-Debugging, Compiler flags.
11 -D_GLIBCXX_DEBUG
12 #define _GLIBCXX_DEBUG
13 // 128-Bit Integer/Float. Muss zum Einlesen/Ausgeben
14 // in einen int oder long long gecastet werden.
15 __int128, __float128
16 // float mit Decimaldarstellung
17 #include <decimal/decimal>
18 std::decimal::decimal128
19 // 1e18 < INF < Max_Value / 2
20 constexpr ll INF = 0x3FFF'FFFF'FFFF'FFFFll;
21 // 1e9 < INF < Max_Value / 2
22 constexpr int INF = 0x3FFF'FFFF;
```

## 7.10 Stress Test

```
1 for i in {1..1000}; do
2     printf "\r%i"
3     python3 gen.py > input # generate test with gen.py
4     ./a.out < input > out # execute ./a.out
5     ./b.out < input > out2 # execute ./b.out
6     diff out out2 || break
7 done
```

## 7.11 Gemischtes

- **(Minimum) Flow mit Demand  $d$ :** Erstelle neue Quelle  $s'$  und Senke  $t'$  und setze die folgenden Kapazitäten:

$$c'(s',v) = \sum_{u \in V} d(u,v) \quad c'(v,t') = \sum_{u \in V} d(v,u)$$

$$c'(u,v) = c(u,v) - d(u,v) \quad c'(t,s) = x$$

Löse Fluss auf  $G'$  mit DINIC's ALGORITHMUS, wenn alle Kanten von  $s'$  saturiert sind ist der Fluss in  $G$  gültig.  $x$  beschränkt den Fluss in  $G$  (Binary-Search für minflow,  $\infty$  sonst).

- **JOHNSONS Reweighting Algorithmus:** Initialisiere alle Entfernungen mit  $d[i] = 0$ . Berechne mit BELLMANN-FORD kürzeste Entfernungen. Falls es einen negativen Zyklus gibt abbrechen. 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 BELLMANN-FORD, um die kürzesten Pfade von  $s$  aus zu finden.  $d[v]$  ist mögliche Lösung für  $v$ .

- **Min-Weight-Vertex-Cover im Bipartiten Graph:** Partitioniere in  $A$ ,  $B$  und füge Kanten  $s \rightarrow A$  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 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 Knoten, die alle Kanten berühren) = Max Matching. Richte Kanten im Matching von  $B$  nach  $A$  und sonst von  $A$  nach  $B$ , makiere alle Knoten die von einem ungematchten Knoten in  $A$  erreichbar sind, das Vertex Cover sind die makierten Knoten aus  $B$  und die unmakierten Knoten aus  $A$ .

- **Bipartites Matching mit Gewichten auf linken Knoten:** Minimiere Matchinggewicht. Lösung: Sortiere Knoten links aufsteigend nach Gewicht, danach nutze normalen Algorithmus (KUHN, Seite 9)

- **Satz von PICK:** Sei  $A$  der Flächeninhalt eines einfachen Gitterpolygons,  $I$  die Anzahl der Gitterpunkte im Inneren und  $R$  die Anzahl der Gitterpunkte auf dem Rand. Es gilt:

$$A = I + \frac{R}{2} - 1$$

- **Lemma von BURNSIDE:** Sei  $G$  eine endliche Gruppe, die auf der Menge  $X$  operiert. Für jedes  $g \in G$  sei  $X^g$  die Menge der Fixpunkte bei Operation durch  $g$ , also  $X^g = \{x \in X \mid g \bullet x = x\}$ . Dann gilt für die Anzahl der Bahnen  $[X/G]$  der Operation:

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

- **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-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.

- **TURAN's-Theorem:** Die Anzahl an Kanten in einem Graphen mit  $n$  Knoten der keine clique der größe  $x+1$  enthält ist:

$$ext(n, K_{x+1}) = \binom{n}{2} - \left[ (x - (n \bmod x)) \cdot \left\lfloor \frac{n}{x} \right\rfloor + (n \bmod x) \cdot \left\lceil \frac{n}{x} \right\rceil \right]$$

- **EULER's-Polyedersatz:** In planaren Graphen gilt  $n - m + f - c = 1$ .
- **PYTHAGOREISCHE TRIPEL:** Sei  $m > n > 0$ ,  $k > 0$  und  $m \not\equiv n \pmod{2}$  dann beschreibt diese Formel alle Pythagoreischen Tripel eindeutig:

$$k \cdot \begin{pmatrix} a = m^2 - n^2, & b = 2mn, & c = m^2 + n^2 \end{pmatrix}$$

- **Centroids of a Tree:** Ein Centroid ist ein Knoten, der einen Baum in Komponenten der maximalen Größe  $\frac{|V|}{2}$  splitted. 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:  $O(|V| \cdot \log(|V|))$ .
- **Gregorian Calendar:** Der Anfangstag des Jahres ist alle 400 Jahre gleich.
- **Pivotsuche und Rekursion auf linkem und rechtem Teilarray:** Suche gleichzeitig von links und rechts nach Pivot, um Worst Case von  $O(n^2)$  zu  $O(n \log n)$  zu verbessern.

- **Mo's Algorithm:** SQRT-Decomposition auf  $n$  Intervall Queries  $[l, r]$ . Gruppier 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.)

- **SQRT Techniques:**

- Aufteilen in *leichte* (wert  $\leq \sqrt{x}$ ) und *schwere* (höchsten  $\sqrt{x}$  viele) Objekte.
- Datenstruktur in Blöcke fester Größe (z.b. 256 oder 512) aufteilen.
- Datenstruktur nach fester Anzahl Updates komplett neu bauen.
- Wenn die Summe über  $x_i$  durch  $X$  beschränkt ist, dann gibt es nur  $\sqrt{2X}$  verschiedene Werte von  $x_i$  (z.b. Längen von Strings).
- Wenn  $w \cdot h$  durch  $X$  beschränkt ist, dann ist  $\min(w, h) \leq \sqrt{X}$ .

- **Partition:** Gegeben Gewichte  $w_0 + w_1 + \dots + w_k = W$ , existiert eine Teilmenge mit Gewicht  $x$ ? Drei gleiche Gewichte  $w$  können zu  $w$  und  $2w$  kombiniert werden ohne die Lösung zu ändern  $\Rightarrow$  nur  $2\sqrt{W}$  unterschiedliche Gewichte. Mit bitsets daher selbst für  $10^5$  lösbar.

## 7.12 Tipps & Tricks

- **Run Time Error:**

- Stack Overflow? Evtl. rekursive Tiefensuche auf langem Pfad?
- Array-Grenzen überprüfen. Indizierung bei 0 oder bei 1 beginnen?
- Abbruchbedingung bei Rekursion?
- Evtl. Memory Limit Exceeded? Mit `/usr/bin/time -v` erhält man den maximalen Speicherverbrauch bei der Ausführung (Maximum resident set size).

- **Strings:**

- Soll "**aa**" kleiner als "**z**" sein oder nicht?
- bit `0x20` beeinflusst Groß-/Kleinschreibung.

- **Zeilenbasierte Eingabe:**

- `getline(cin, str)` liest Zeile ein.
- Wenn vorher `cin >> ...` benutzt, lese letztes `\n` mit `getline(cin, x)`.

- **Gleitkommazahlen:**

- NaN? Evtl. ungültige Werte für mathematische Funktionen, z.B. `acos(1.000000000000001)?`
- Falsches Runden bei negativen Zahlen? Abschneiden  $\neq$  Abrunden!
- genügend Präzision oder 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.
- Ausgabeformat im 'unmöglich'-Fall überprüfen.
- Ist das Ergebnis modulo einem Wert?
- Integer Division rundet zur 0  $\neq$  abrunden.
- Eingabegrößen überprüfen. Sonderfälle ausprobieren.
  - $n=0$ ,  $n=-1$ ,  $n=1$ ,  $n=2^{31}-1$ ,  $n=-2^{31}$
  - $n$  gerade/ungerade
  - Graph ist leer/enthält nur einen Knoten.
  - Liste ist leer/enthält nur ein Element.
  - Graph ist Multigraph (enthält Schleifen/Mehrfachkanten).
  - Sind Kanten gerichtet/ungerichtet?
  - Kolineare Punkte existieren.
  - Polygon ist konkav/selbstschneidend.
- Bei DP/Rekursion: Stimmt Basisfall?
- Unsicher bei benutzten STL-Funktionen?

## 8 Template

### 8.1 C++

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define tsolve int t; cin >> t; while(t--) solve
4 #define all(x) ::begin(x), ::end(x)
5 #define sz(x) (ll)::size(x)
6 using ll = long long;
7 using ld = long double;
8 void solve() {}
9 int main() {
10     cin.tie(0) -> sync_with_stdio(false);
11     cout << setprecision(16);
12     solve();
13 }
```

### 8.2 Console

```
1 alias comp=
2   "g++ -std=gnu++17 -O2 -Wall -Wextra -Wconversion -Wshadow"
3 alias dbg="comp -g -fsanitize=address,undefined"
```



## 9 Tests

Dieser Abschnitt enthält lediglich Dinge die während der Practicesession getestet werden sollten!

### 9.1 GCC

- sind c++14 Feature vorhanden?
- sind c++17 Feature vorhanden?
- kompiliert dieser Code:

```
1 //https://gcc.gnu.org/bugzilla/show_bug.cgi?id=68203
2 struct A {
3     pair<int, int> values[1000000];
4 };
```

- funktioniert `__int128`?
- funktionieren Pragas?
- funktionieren `constexpr` zur Compilezeit (+Zeitlimit)?
- wie groß ist `sizeof(char*)`?
- wie groß ist `RAND_MAX`?
- funktioniert `random_device`? (und gib es unterschiedliche Ergebnisse?)
- funktioniert `clock()`?

### 9.2 Python

- Rekursionslimit?

### 9.3 Judge

- ist der Checker casesensitive?
- wie werden zusätzliches Whitespacecharacter bei sonst korrektem Output behandelt?
- vergleiche ausführungszeit auf dem judge und lokal (z.b. mit Primzahl Sieb)

```
1 "\r\r\r\n\t \r\n\r"
```

### 9.4 Precision

- Mode 0 means no excess precision
- Mode 2 means excess precision (all operations in 80 bit floats)
- Result 0 without excess precision (expected floating point error)
- $\sim 8e^{-17}$  with excess precision (real value)

```
1 #include <cfloat>
2
3 int main() {
4     cout << "Mode: " << FLT_EVAL_METHOD << endl;
5     double a = atof("1.2345678");
6     double b = a*a;
7     cout << b - 1.52415765279683990130 << '\n';
8 }
```