1 Datenstrukturen

1.1 Union-Find

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

```
// unions[i] >= 0 => unions[i] =  parent
// unions[i]  < 0 => unions[i] = -height
vector<int> unions;

void init(int n) { //Initialisieren
  unions.assign(n, -1);
}

int findSet(int n) { // Pfadkompression
  if (unions[n] < 0) return n;
  return unions[n] = findSet(unions[n]);
}

void linkSets(int a, int b) { // Union by rank.
  if (unions[a] > unions[b]) unions[a] = b;
  else if (unions[b] > unions[a]) unions[b] = a;
  else {
    unions[a] = b;
    unions[b]--;
}}

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

1.2 Segmentbaum

`init`	baut den Baum auf	$\mathcal{O}(n)$
`query`	findet das min(max) in [l, r)	$\mathcal{O}(\log(n))$
`update`	ändert einen Wert	$\mathcal{O}(\log(n))$

```
vector<ll> tree;
constexpr ll neutral = 0; // Neutral element for combine

ll combine(ll a, ll b) {
  return a + b;
}

void init(vector<ll>& a) {
  tree.assign(2 * sz(a), 0);
  copy(all(a), tree.begin() + sz(a));
  for (int i = sz(tree)/2 - 1; i > 0; i--) {
    tree[i] = combine(tree[2 * i], tree[2 * i + 1]);
}}

void update(int i, ll val) {
  for (tree[i += sz(tree)/2] = val; i /= 2; ) {
    tree[i] = combine(tree[2 * i], tree[2 * i + 1]);
}}

ll query(int l, int r) {
  ll resL = neutral, resR = neutral;
  for (l += sz(tree)/2, r += sz(tree)/2; l < r; l /= 2, r /= 2) {
    if (l&1) resL = combine(resL, tree[l++]);
    if (r&1) resR = combine(tree[--r], resR);
  }
  return combine(resL, resR);
}

// Oder: Intervall-Modifikation, Punkt-Query:
void modify(int l, int r, ll val) {
  for (l += sz(tree)/2, r += sz(tree)/2; l < r; l /= 2, r /= 2) {
    if (l&1) {tree[l] = combine(tree[l], val); l++;};
    if (r&1) {--r; tree[r] = combine(tree[r], val);};
}}

ll query(int i) {
  ll res = neutral;
  for (i += sz(tree)/2; i > 0; i /= 2) {
    res = combine(res, tree[i]);
  }
  return res;
}
```

1.2.1 Lazy Propagation

Assignment modifications, sum queries

`lower_bound` erster Index in [l, r) ≥ x (erfordert max-combine) $\mathcal{O}(\log(n))$

```
struct SegTree {
  int size, height;
  static constexpr ll neutral = 0; // Neutral element for combine
  static constexpr ll updateFlag = 0; // Unused value by updates
  vector<ll> tree, lazy;

  SegTree(const vector<ll>& a) : SegTree(sz(a)) {
    copy(all(a), tree.begin() + size);
    for (int i = size - 1; i > 0; i--)
      tree[i] = combine(tree[2 * i], tree[2 * i + 1]);
  }

  SegTree(int n) : size(n), height(__lg(2 * n)),
    tree(2 * n, neutral), lazy(n, updateFlag) {}

  ll combine(ll a, ll b) {return a + b;} // Modify this + neutral

  void apply(int i, ll val, int k) { // And this + updateFlag
    tree[i] = val * k;
    if (i < size) lazy[i] = val; // Don't forget this
  }

  void push_down(int i, int k) {
    if (lazy[i] != updateFlag) {
      apply(2 * i, lazy[i], k);
      apply(2 * i + 1, lazy[i], k);
      lazy[i] = updateFlag;
  }}

  void push(int i) {
    for (int s = height, k = 1 << (height-1); s > 0; s--, k /= 2)
      push_down(i >> s, k);
  }

  void build(int i) {
    for (int k = 2; i /= 2; k *= 2) {
      push_down(i, k / 2);
      tree[i] = combine(tree[2 * i], tree[2 * i + 1]);
  }}

  void update(int l, int r, ll val) { // data[l..r) = val
    l += size, r += size;
    int l0 = l, r0 = r;
    push(l0), push(r0 - 1);
    for (int k = 1; l < r; l /= 2, r /= 2, k *= 2) {
      if (l&1) apply(l++, val, k);
      if (r&1) apply(--r, val, k);
    }
    build(l0), build(r0 - 1);
  }

  ll query(int l, int r) { // sum[l..r)
    l += size, r += size;
    push(l), push(r - 1);
    ll resL = neutral, resR = neutral;
    for (; l < r; l /= 2, r /= 2) {
      if (l&1) resL = combine(resL, tree[l++]);
      if (r&1) resR = combine(tree[--r], resR);
    }
    return combine(resL, resR);
  }

  // Optional:
  ll lower_bound(int l, int r, int x) {
    l += size, r += size;
    push(l), push(r - 1);
    vector<pair<int, int>> a, st;
    for (int k = 1; l < r; l /= 2, r /= 2, k *= 2) {
      if (l&1) a.emplace_back(l++, k);
      if (r&1) st.emplace_back(--r, k);
    }
    a.insert(a.end(), st.rbegin(), st.rend());
    for (auto [i, k] : a) {
      if (tree[i] >= x) return find(i, x, k); // Modify this
    }
    return -1;
  }

  ll find(int i, int x, int k) {
    if (i >= size) return i - size;
    push_down(i, k / 2);
    if (tree[2*i] >= x) return find(2 * i, x, k / 2); // And this
    else return find(2 * i + 1, x, k / 2);
  }
};
```

1.3 STL-Bitset

```
bitset<10> bits(0b000010100);
cout << bits._Find_first() << endl; //2
cout << bits._Find_next(2) << endl; //4
cout << bits._Find_next(4) << endl; //10 bzw. N
bits[x] = 1;    //not bits.set(x)!
bits[x] = 0;    //not bits.reset(x)!
bits[x].flip(); //not bits.flip(x)!
```

1.4 Fenwick Tree

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

```
vector<ll> tree;
void update(int i, ll val) {
  for (i++; i < sz(tree); i += (i & (-i))) tree[i] += val;
}
void init(int n) {
  tree.assign(n + 1,0);
}
ll prefix_sum(int i) {
  ll sum = 0;
  for (i++; i > 0; i -= (i & (-i))) sum += tree[i];
  return sum;
}
```

init	baut den Baum auf	$\mathcal{O}(n \cdot \log(n))$
prefix_sum	summe von [0, i]	$\mathcal{O}(\log(n))$
update	addiert ein Delta zu allen Elementen [l, r)	$\mathcal{O}(\log(n))$

```
vector<ll> add, mul;
void update(int l, int r, ll val) {
  for (int tl = l + 1; tl < sz(add); tl += tl&(-tl))
    add[tl] += val, mul[tl] -= val * l;
  for (int tr = r + 1; tr < sz(add); tr += tr&(-tr))
    add[tr] -= val, mul[tr] += val * r;
}
void init(vector<ll>& v) {
  mul.assign(sz(v) + 1,0);
  add.assign(sz(v) + 1,0);
  for(int i = 0; i < sz(v); i++) update(i, i + 1, v[i]);
}
ll prefix_sum (int i) {
  ll res = 0; i++;
  for (int ti = i; ti > 0; ti -= ti&(-ti))
    res += add[ti] * i + mul[ti];
  return res;
}
```

1.5 Wavelet Tree

Constructor	baut den Baum auf	$\mathcal{O}(n \cdot \log(n))$
kth	sort $[l,r)[k]$	$\mathcal{O}(\log(n))$
countSmaller	Anzahl elemente in $[l,r)$ kleiner als k	$\mathcal{O}(\log(n))$

```
struct WaveletTree {
  using it = vector<ll>::iterator;
  WaveletTree *ln, *rn;
  ll lo, hi;
  vector<int> b;
private:
  WaveletTree(it from, it to, ll x, ll y)
  : ln(nullptr), rn(nullptr), lo(x), hi(y), b(1) {
    ll mid = (lo + hi) / 2;
    auto f = [&](ll x){return x < mid;};
    for (it c = from; c != to; c++) {
      b.push_back(b.back() + f(*c));
    }
    if (lo + 1 >= hi || from == to) return;
    it pivot = stable_partition(from, to, f);
    ln = new WaveletTree(from, pivot, lo, mid);
    rn = new WaveletTree(pivot, to, mid, hi);
  }
public:
  WaveletTree(vector<ll> in) : WaveletTree(all(in),
    *min_element(all(in)), *max_element(all(in)) + 1){}
  // kth element in sort[l, r) all 0-indexed
  ll kth(int l, int r, int k) {
    if (l >= r || k >= r - l) return -1;
    if (lo + 1 >= hi) return lo;
    int inLeft = b[r] - b[l];
    if (k < inLeft) {
      return ln->kth(b[l], b[r], k);
    } else {
      return rn->kth(l-b[l], r-b[r], k-inLeft);
  }}
  // count elements in[l, r) smaller than k
  int countSmaller(int l, int r, ll k) {
    if (l >= r || k <= lo) return 0;
    if (hi <= k) return r - l;
    return ln->countSmaller(b[l], b[r], k) +
           rn->countSmaller(l-b[l], r-b[r], k);
  }
  ~WaveletTree(){
    delete ln;
    delete rn;
  }
};
```

1.6 (Implicit) Treap (Cartesian Tree)

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

```
mt19937 rng(0xc4bd5dad);
struct Treap {
  struct Node {
    ll val;
    int prio, size = 1, l = -1, r = -1;
    Node (ll x) : val(x), prio(rng()) {}
  };
  vector<Node> treap;
  int root = -1;
  int getSize(int v) {
    return v < 0 ? 0 : treap[v].size;
  }
  void upd(int v) {
    if (v < 0) return;
    auto *V = &treap[v];
    V->size = 1 + getSize(V->l) + getSize(V->r);
    // Update Node Code
  }
  void push(int v) {
    if (v < 0) return;
    //auto *V = &treap[v];
    //if (V->lazy) {
    //  Lazy Propagation Code
    //  if (V->l >= 0) treap[V->l].lazy = true;
    //  if (V->r >= 0) treap[V->r].lazy = true;
    //  V->lazy = false;
    //}
  }
  pair<int, int> split(int v, int k) {
    if (v < 0) return {-1, -1};
    auto *V = &treap[v];
    push(v);
    if (getSize(V->l) >= k) { // "V->val >= k" for lower_bound(k)
      auto [left, right] = split(V->l, k);
      V->l = right;
      upd(v);
      return {left, v};
    } else {
      // and only "k"
      auto [left, right] = split(V->r, k - getSize(V->l) - 1);
      V->r = left;
      upd(v);
      return {v, right};
  }}
  int merge(int left, int right) {
    if (left < 0) return right;
    if (right < 0) return left;
    if (treap[left].prio < treap[right].prio) {
      push(left);
      treap[left].r = merge(treap[left].r, right);
      upd(left);
      return left;
    } else {
      push(right);
      treap[right].l = merge(left, treap[right].l);
      upd(right);
      return right;
  }}
  void insert(int i, ll val) { // and i = val
    auto [left, right] = split(root, i);
    treap.emplace_back(val);
    left = merge(left, sz(treap) - 1);
    root = merge(left, right);
  }
  void remove(int i, int count = 1) {
    auto [left, t_right] = split(root, i);
    auto [middle, right] = split(t_right, count);
    root = merge(left, right);
  }
  // for query use remove and read middle BEFORE remerging
};
```

1.7 Range Minimum Query

init	baut Struktur auf	$\mathcal{O}(n \cdot \log(n))$
queryIdempotent	Index des Minimums in [l, r)	$\mathcal{O}(1)$

- better-Funktion muss idempotent sein!

```cpp
struct SparseTable {
  vector<vector<int>> st;
  vector<ll> *a;

  int better(int lidx, int ridx) {
    return a->at(lidx) <= a->at(ridx) ? lidx : ridx;
  }

  void init(vector<ll> *vec) {
    a = vec;
    st.assign(__lg(sz(*a)) + 1, vector<int>(sz(*a)));
    iota(all(st[0]), 0);
    for (int j = 0; (2 << j) <= sz(*a); j++) {
      for (int i = 0; i + (2 << j) <= sz(*a); i++) {
        st[j + 1][i] = better(st[j][i] , st[j][i + (1 << j)]);
  }}}

  int queryIdempotent(int l, int r) {
    int j = __lg(r - l); //31 - builtin_clz(r - l);
    return better(st[j][l] , st[j][r - (1 << j)]);
  }
};
```

1.8 STL-Tree

```cpp
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std; using namespace __gnu_pbds;
template<typename T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
                  tree_order_statistics_node_update>;

int main() {
  Tree<int> X;
  // insert {1, 2, 4, 8, 16}
  for (int i = 1; i <= 16; i *= 2) X.insert(i);
  cout << *X.find_by_order(3) << endl; // => 8
  cout << X.order_of_key(10) << endl;
  // => 4 = min i, mit X[i] >= 10
  return 0;
}
```

1.9 STL-Rope (Implicit Cartesian Tree)

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

1.10 STL HashMap

3 bis 4 mal langsamer als `std::vector` aber 8 bis 9 mal schneller als `std::map`

```cpp
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

template<typename T>
struct betterHash {
  size_t operator()(T o) const {
    size_t h = hash<T>()(o) ^ 42394245; //random value
    h = ((h >> 16) ^ h) * 0x45d9f3b;
    h = ((h >> 16) ^ h) * 0x45d9f3b;
    h = ((h >> 16) ^ h);
    return h;
}};

template<typename K, typename V, typename H = betterHash<K>>
using hashMap = gp_hash_table<K, V, H>;
template<typename K, typename H = betterHash<K>>
using hashSet = gp_hash_table<K, null_type, H>;
```

1.11 STL Priority Queue

Nicht notwendig, wenn Smaller-Larger-Optimization greift.

```cpp
#include <ext/pb_ds/priority_queue.hpp>
template<typename T>
// greater<T> für Min-Queue
using priorityQueue = __gnu_pbds::priority_queue<T, less<T>>;

int main() {
  priorityQueue<int> pq;
  auto it = pq.push(5); // O(1)
  pq.push(7);
  pq.pop(); // O(log n) amortisiert
  pq.modify(it, 6); // O(log n) amortisiert
  pq.erase(it); // O(log n) amortisiert
  priorityQueue<int> pq2;
  pq.join(pq2); // O(1)
}
```

1.12 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.

```cpp
// Lower Envelope mit MONOTONEN Inserts und Queries. Jede neue
// Gerade hat kleinere Steigung als alle vorherigen.
vector<ll> ms, bs; int ptr = 0;

bool bad(int l1, int l2, int l3) {
  return (bs[l3]-bs[l1])*(ms[l1]-ms[l2]) <
         (bs[l2]-bs[l1])*(ms[l1]-ms[l3]);
}

void add(ll m, ll b) { // Laufzeit O(1) amortisiert
  ms.push_back(m); bs.push_back(b);
  while (sz(ms) >= 3 && bad(sz(ms)-3, sz(ms)-2, sz(ms)-1)) {
    ms.erase(ms.end() - 2); bs.erase(bs.end() - 2);
  }
  ptr = min(ptr, sz(ms) - 1);
}

ll get(int idx, ll x) {return ms[idx] * x + bs[idx];}

ll query(ll x) { // Laufzeit: O(1) amortisiert
  if (ptr >= sz(ms)) ptr = sz(ms) - 1;
  while (ptr < sz(ms)-1 && get(ptr + 1, x) < get(ptr, x)) ptr++;
  return get(ptr, x);
}
```

```cpp
struct Line {
  mutable ll m, b, p;
  bool operator<(const Line& o) const {return m < o.m;}
  bool operator<(ll x) const {return p < x;}
};

struct HullDynamic : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static constexpr ll INF = LLONG_MAX;
  ll div(ll a, ll b) {return a / b - ((a ^ b) < 0 && a % b);}

  bool isect(iterator x, iterator y) {
    if (y == end()) {x->p = INF; return false;}
    if (x->m == y->m) x->p = x->b > y->b ? INF : -INF;
    else x->p = div(y->b - x->b, x->m - y->m);
    return x->p >= y->p;
  }

  void add(ll m, ll b) {
    auto x = insert({m, b, 0});
    while (isect(x, next(x))) erase(next(x));
    if (x != begin()) {
      x--;
      if (isect(x, next(x))) {
        erase(next(x));
        isect(x, next(x));
    }}
    while (x != begin() && prev(x)->p >= x->p) {
      x--;
      isect(x, erase(next(x)));
  }}

  ll query(ll x) {
    auto l = *lower_bound(x);
    return l.m * x + l.b;
  }
};
```

1.13 Link-Cut-Tree

`Constructor`	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	$O(\log(n))$

```
constexpr ll queryDefault = 0;
constexpr ll updateDefault = 0;
ll _modify(ll x, ll y) {
  return x + y;
}
ll _query(ll x, ll y) {
  return x + y;
}
ll _update(ll delta, int length) {
  if (delta == updateDefault) return updateDefault;
  //ll result = delta
  //for (int i=1; i<length; i++) result = _query(result, delta);
  return delta * length;
}
//generic:
ll joinValueDelta(ll value, ll delta) {
  if (delta == updateDefault) return value;
  return _modify(value, delta);
}
ll joinDeltas(ll delta1, ll delta2) {
  if (delta1 == updateDefault) return delta2;
  if (delta2 == updateDefault) return delta1;
  return _modify(delta1, delta2);
}
struct LCT {
  struct Node {
    ll nodeValue, subTreeValue, delta;
    bool revert;
    int id, size;
    Node *left, *right, *parent;
    Node(int id = 0, int val = queryDefault) :
      nodeValue(val), subTreeValue(val), delta(updateDefault),
      revert(false), id(id), size(1),
      left(nullptr), right(nullptr), parent(nullptr) {}
    bool isRoot() {
      return !parent || (parent->left != this &&
             parent->right != this);
    }
    void push() {
      if (revert) {
        revert = false;
        swap(left, right);
        if (left) left->revert ^= 1;
        if (right) right->revert ^= 1;
      }
      nodeValue = joinValueDelta(nodeValue, delta);
      subTreeValue = joinValueDelta(subTreeValue,
                                    _update(delta, size));
      if (left) left->delta = joinDeltas(left->delta, delta);
      if (right) right->delta = joinDeltas(right->delta, delta);
      delta = updateDefault;
    }
    ll getSubtreeValue() {
      return joinValueDelta(subTreeValue, _update(delta, size));
    }
    void update() {
      subTreeValue = joinValueDelta(nodeValue, delta);
      size = 1;
      if (left) {
        subTreeValue = _query(subTreeValue,
                              left->getSubtreeValue());
        size += left->size;
      }
      if (right) {
        subTreeValue = _query(subTreeValue,
                              right->getSubtreeValue());
        size += right->size;
    }}
  };
  vector<Node> nodes;
  LCT(int n) : nodes(n) {
    for (int i = 0; i < n; i++) nodes[i].id = i;
  }
  void connect(Node* ch, Node* p, int isLeftChild) {
    if (ch) ch->parent = p;
    if (isLeftChild >= 0) {
      if (isLeftChild) p->left = ch;
      else p->right = ch;
  }}
  void rotate(Node* x) {
    Node* p = x->parent;
    Node* g = p->parent;
    bool isRootP = p->isRoot();
    bool leftChildX = (x == p->left);
    connect(leftChildX ? x->right : x->left, p, leftChildX);
    connect(p, x, !leftChildX);
    connect(x, g, isRootP ? -1 : p == g->left);
    p->update();
  }
  void splay(Node* x) {
    while (!x->isRoot()) {
      Node* p = x->parent;
      Node* g = p->parent;
      if (!p->isRoot()) g->push();
      p->push();
      x->push();
      if (!p->isRoot()) rotate((x == p->left) ==
                               (p == g->left) ?  p : x);
      rotate(x);
    }
    x->push();
    x->update();
  }
  Node* expose(Node* x) {
    Node* last = nullptr;
    for (Node* y = x; y; y = y->parent) {
      splay(y);
      y->left = last;
      last = y;
    }
    splay(x);
    return last;
  }
  void makeRoot(Node* x) {
    expose(x);
    x->revert ^= 1;
  }
  bool connected(Node* x, Node* y) {
    if (x == y) return true;
    expose(x);
    expose(y);
    return x->parent;
  }
  void link(Node* x, Node* y) {
    assert(!connected(x, y)); // not yet connected!
    makeRoot(x);
    x->parent = y;
  }
  void cut(Node* x, Node* y) {
    makeRoot(x);
    expose(y);
    //must be a tree edge!
    assert(!(y->right != x || x->left != nullptr));
    y->right->parent = nullptr;
    y->right = nullptr;
  }
  Node* lca(Node* x, Node* y) {
    assert(connected(x, y));
    expose(x);
    return expose(y);
  }
  ll query(Node* from, Node* to) {
    makeRoot(from);
    expose(to);
    if (to) return to->getSubtreeValue();
    return queryDefault;
  }
  void modify(Node* from, Node* to, ll delta) {
    makeRoot(from);
    expose(to);
    to->delta = joinDeltas(to->delta, delta);
  }
};
```

1.14 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	$O(1)$

```cpp
template<typename T>
struct persistent {
  int& time;
  vector<pair<int, T>> data;

  persistent(int& time, T value = {})
    : time(time), data(1, {time, value}) {}

  T get(int t) {
    return prev(upper_bound(all(data),
                            pair<int, T>(t+1, {})))->second;
  }

  int set(T value) {
    time+=2;
    data.push_back({time, value});
    return time;
  }
};
```

```cpp
template<typename T>
struct persistentArray{
  int time = 0;
  vector<persistent<T>> data;
  vector<pair<int, int>> mods;

  persistentArray(int n, T value = {})
    : time(0), data(n, {time, value}) {}

  T get(int p, int t) {
    return data[p].get(t);
  }

  int set(int p, T value) {
    mods.push_back({p, time});
    return data[p].set(value);
  }

  void reset(int t) {
    while (!mods.empty() && mods.back().second > t) {
      data[mods.back().first].data.pop_back();
      mods.pop_back();
    }
    time = t;
  }
};
```

2 Graphen

2.1 DFS

Kantentyp (v,w)	`dfs[v] < dfs[w]`	`fin[v] > fin[w]`	`seen[w]`
in-tree	`true`	`true`	`false`
forward	`true`	`true`	`true`
backward	`false`	`false`	`true`
cross	`false`	`true`	`true`

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 Spannbaums.

2.3 Kruskal

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

```cpp
sort(all(edges));
vector<edge> mst;
int cost = 0;
for (edge& e : edges) {
  if (findSet(e.from) != findSet(e.to)) {
    unionSets(e.from, e.to);
    mst.push_back(e);
    cost += e.cost;
}}
```

2.4 Erdős-Gallai

Sei $d_1 \geq \cdots \geq d_n$. Es existiert genau dann ein Graph G mit Degreesequence d falls $\sum\limits_{i=1}^{n} d_i$ gerade ist und für $1 \leq k \leq n$: $\sum\limits_{i=1}^{k} d_i \leq k\cdot(k-1) + \sum\limits_{i=k+1}^{n} \min(d_i, k)$

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

```cpp
vector<vector<int>> havelHakimi(const vector<int>& deg) {
  priority_queue<pair<int, int>> pq;
  for (int i = 0; i < sz(deg); i++) pq.push({deg[i], i});
  vector<vector<int>> adj;
  while (!pq.empty()) {
    auto [degV, v] = pq.top(); pq.pop();
    if (sz(pq) < degV) return {}; //impossible
    vector<pair<int, int>> todo;
    for (int i = 0; i < degV; i++) {
      auto [degU, v] = pq.top(); pq.pop();
      adj[v].push_back(u);
      adj[u].push_back(v);
      if (degU > 1) todo.push_back({degU - 1, u});
    }
    for (auto e : todo) pq.push(e);
  }
  return adj;
}
```

2.5 Centroids

`find_centroid` findet alle Centroids des Baums (maximal 2) $O(|V|)$

```cpp
vector<int> s;
void dfs1(int u, int v = -1) {
  s[u] = 1;
  for (int w : adj[u]) {
    if (w == v) continue;
    dfs1(w, u);
    s[u] += s[w];
}}

pair<int, int> dfs2(int u, int v, int n) {
  for (int w : adj[u]) {
    if (2 * s[w] == n) return {u, w};
    if (w != v && 2 * s[w] > n) return dfs2(w, u, n);
  }
  return {u, -1};
}

pair<int, int> find_centroid(int root) {
  // s muss nicht initialisiert werden, nur groß genug sein
  dfs1(root);
  return dfs2(root, -1, s[root]);
}
```

2.6 Baum-Isomorphie

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

```cpp
vector<vector<int>> adj;
map<vector<int>, int> known;

int treeLabel(int root, int p = -1) {
  vector<int> children;
  for (int x : adj[root]) {
    if (x == p) continue;
    children.push_back(treeLabel(x, root));
  }
  sort(all(children));
  if (known.find(children) == known.end()) {
    known[children] = sz(known);
  }
  return known[children];
}
```

2.7 Kürzeste Wege

2.7.1 Algorithmus von Dijkstra

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

```
using path = pair<ll, int>; //dist, destination

void dijkstra(const vector<vector<path>> &adjlist, int start) {
  priority_queue<path, vector<path>, greater<path>> pq;
  vector<ll> dist(sz(adjlist), INF);
  vector<int> prev(sz(adjlist), -1);
  dist[start] = 0; pq.emplace(0, start);

  while (!pq.empty()) {
    auto [dc, c] = pq.top(); pq.pop();
    if (dc > dist[c]) continue; // WICHTIG!

    for (auto [dx, x] : adjlist[c]) {
      ll newDist = dc + dx;
      if (newDist < dist[x]) {
        dist[x] = newDist;
        prev[x] = c;
        pq.emplace(newDist, x);
  }}}
  //return dist, prev;
}
```

2.7.2 Bellmann-Ford-Algorithmus

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

```
void bellmannFord(int n, vector<edge> edges, int start) {
  vector<ll> dist(n, INF), parent(n, -1);
  dist[start] = 0;

  for (int i = 1; i < n; i++) {
    for (edge& e : edges) {
      if (dist[e.from] != INF &&
          dist[e.from] + e.cost < dist[e.to]) {
        dist[e.to] = dist[e.from] + e.cost;
        parent[e.to] = e.from;
  }}}

  for (edge& e : edges) {
    if (dist[e.from] != INF &&
        dist[e.from] + e.cost < dist[e.to]) {
      // Negativer Kreis gefunden.
  }}
  //return dist, parent;
}
```

2.7.3 Floyd-Warshall-Algorithmus

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

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

```
vector<vector<ll>> dist; // Entfernung zwischen je zwei Punkten.
vector<vector<int>> pre;

void floydWarshall() {
  pre.assign(sz(dist), vector<int>(sz(dist), -1));
  for (int i = 0; i < sz(dist); i++) {
    for (int j = 0; j < sz(dist); j++) {
      if (dist[i][j] < INF) {
        pre[i][j] = j;
  }}}

  for (int k = 0; k < sz(dist); k++) {
    for (int i = 0; i < sz(dist); i++) {
      for (int j = 0; j < sz(dist); j++) {
        if (dist[i][j] > dist[i][k] + dist[k][j]) {
          dist[i][j] = dist[i][k] + dist[k][j];
          pre[i][j] = pre[i][k];
}}}}}

vector<int> getPath(int u, int v) {
  //return dist[u][v]; // Pfadlänge u -> v
  if (pre[u][v] < 0) return {};
  vector<int> path = {v};
  while (u != v) path.push_back(u = pre[u][v]);
  return path; //Pfad u -> v
}
```

2.7.4 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: $c_{ij} = a_{ij} * b_{ij} = \min\{a_{ik} + b_{kj}\}$

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: $c_{ij} = a_{ij} \cdot b_{ij} = \sum a_{ik} + b_{kj}$

2.8 Lowest Common Ancestor

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

```
struct LCA {
  vector<ll> depth;
  vector<int> visited, first;
  int idx;
  SparseTable st; //sparse table von oben

  void init(vector<vector<int>>& g, int root) {
    depth.assign(2 * sz(g), 0);
    visited.assign(2 * sz(g), -1);
    first.assign(sz(g), 2 * sz(g));
    idx = 0;
    visit(g, root);
    st.init(&depth);
  }

  void visit(vector<vector<int>>& g, int v, ll d=0, int p=-1) {
    visited[idx] = v, depth[idx] = d;
    first[v] = min(idx, first[v]), idx++;

    for (int w : g[v]) {
      if (first[w] == 2 * sz(g)) {
        visit(g, w, d + 1, v);
        visited[idx] = v, depth[idx] = d, idx++;
  }}}

  int getLCA(int a, int b) {
    if (first[a] > first[b]) swap(a, b);
    return visited[st.queryIdempotent(first[a], first[b] + 1)];
  }

  ll getDepth(int a) {return depth[first[a]];}
};
```

2.9 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])`.

```
vector<vector<int>> adj;
vector<int> sz, in, out, nxt, par;
int t;

void dfs_sz(int v = 0, int from = -1) {
  sz[v] = 1;
  for (auto& u : adj[v]) {
    if (u != from) {
      dfs_sz(u, v);
      sz[v] += sz[u];
    }
    if (adj[v][0] == from || sz[u] > sz[adj[v][0]]) {
      swap(u, adj[v][0]);
}}}

void dfs_hld(int v = 0, int from = -1) {
  par[v] = from;
  in[v] = t++;
  for (int u : adj[v]) {
    if (u == from) continue;
    nxt[u] = (u == adj[v][0] ? nxt[v] : u);
    dfs_hld(u, v);
  }
  out[v] = t;
}

void init() {
  int n = sz(adj);
  sz.assign(n, 0); in.assign(n, 0); out.assign(n, 0);
  nxt.assign(n, 0); par.assign(n, -1);
  t = 0;
  dfs_sz(); dfs_hld();
}

vector<pair<int, int>> get_intervals(int u, int v) {
  vector<pair<int, int>> res;
  while (true) {
    if (in[v] < in[u]) swap(u, v);
    if (in[nxt[v]] <= in[u]) {
      res.emplace_back(in[u], in[v] + 1);
      return res;
    }
    res.emplace_back(in[nxt[v]], in[v] + 1);
    v = par[nxt[v]];
}}

int get_lca(int u, int v) {
  while (true) {
    if (in[v] < in[u]) swap(u, v);
    if (in[nxt[v]] <= in[u]) return in[u];
    v = par[nxt[v]];
}}
```

2.10 Maximal Cliques

`bronKerbosch` berechnet alle maximalen Cliquen $\mathcal{O}(3^{\frac{n}{3}})$
`addEdge` fügt **ungerichtete** Kante ein $\mathcal{O}(1)$

```
using bits = bitset<64>;
vector<bits> adj, cliques;

void addEdge(int a, int b) {
  if (a != b) adj[a][b] = adj[b][a] = 1;
}

void bronKerboschRec(bits R, bits P, bits X) {
  if (!P.any() && !X.any()) {
    cliques.push_back(R);
  } else {
    int q = (P | X)._Find_first();
    bits cands = P & ~adj[q];
    for (int i = 0; i < sz(adj); i++) if (cands[i]){
      R[i] = 1;
      bronKerboschRec(P & adj[i], X & adj[i], R);
      R[i] = P[i] = 0;
      X[i] = 1;
}}}

void bronKerbosch() {
  cliques.clear();
  bronKerboschRec({}, {(1ull << sz(adj)) - 1}, {});
}
```

2.11 Artikulationspunkte, Brücken und BCC

`find` berechnet Artikulationspunkte, Brücken und BCC $\mathcal{O}(|V|+|E|)$
Wichtig: isolierte Knoten und Brücken sind keine BCC.

```
vector<vector<edge>> adjlist;
vector<int> num;
int counter, rootCount, root;
vector<bool> isArt;
vector<edge> bridges, st;
vector<vector<edge>> bcc;

int dfs(int v, int parent = -1) {
  int me = num[v] = ++counter, top = me;
  for (edge& e : adjlist[v]) {
    if (e.id == parent){}
    else if (num[e.to]) {
      top = min(top, num[e.to]);
      if (num[e.to] < me) st.push_back(e);
    } else {
      if (v == root) rootCount++;
      int si = sz(st);
      int up = dfs(e.to, e.id);
      top = min(top, up);
      if (up >= me) isArt[v] = true;
      if (up > me) bridges.push_back(e);
      if (up <= me) st.push_back(e);
      if (up == me) {
        bcc.emplace_back();
        while (sz(st) > si) {
          bcc.back().push_back(st.back());
          st.pop_back();
  }}}}
  return top;
}

void find() {
  counter = 0;
  num.assign(sz(adjlist), 0);
  isArt.assign(sz(adjlist), false);
  bridges.clear();
  st.clear();
  bcc.clear();
  for (int v = 0; v < sz(adjlist); v++) {
    if (!num[v]) {
      root = v;
      rootCount = 0;
      dfs(v);
      isArt[v] = rootCount > 1;
}}}
```

2.12 Strongly Connected Components (Tarjan)

`scc` berechnet starke Zusammenhangskomponenten $\mathcal{O}(|V|+|E|)$

```
vector<vector<int>> adjlist;

int counter, sccCounter;
vector<bool> inStack;
vector<vector<int>> sccs;
// idx enthält den Index der SCC pro Knoten.
vector<int> d, low, idx, s;

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

  for (auto u : adjlist[v]) {
    if (d[u] < 0) {
      visit(u);
      low[v] = min(low[v], low[u]);
    } else if (inStack[u]) {
      low[v] = min(low[v], low[u]);
  }}

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

void scc() {
  inStack.assign(sz(adjlist), false);
  d.assign(sz(adjlist), -1);
  low.assign(sz(adjlist), -1);
  idx.assign(sz(adjlist), -1);

  counter = sccCounter = 0;
  for (int i = 0; i < sz(adjlist); i++) {
    if (d[i] < 0) visit(i);
}}
```

2.13 2-SAT

```
struct sat2 {
  int n; // + scc variablen
  vector<int> sol;

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

  static int var(int i) {return i << 1;} // use this!

  void addImpl(int a, int b) {
    adjlist[a].push_back(b);
    adjlist[1^b].push_back(1^a);
  }
  void addEquiv(int a, int b) {addImpl(a, b); addImpl(b, a);}
  void addOr(int a, int b) {addImpl(1^a, b);}
  void addXor(int a, int b) {addOr(a, b); addOr(1^a, 1^b);}
  void addTrue(int a) {addImpl(1^a, a);}
  void addFalse(int a) {addTrue(1^a);}
  void addAnd(int a, int b) {addTrue(a); addTrue(b);}
  void addNand(int a, int b) {addOr(1^a, 1^b);}

  bool solvable() {
    scc(); //scc code von oben
    for (int i = 0; i < n; i += 2) {
      if (idx[i] == idx[i + 1]) return false;
    }
    return true;
  }

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

2.14 Global Mincut

`stoer_wagner` berechnet globalen Mincut $\mathcal{O}(|V|^2 \cdot \log(|E|))$
`merge(a,b)` merged Knoten b in Knoten a $\mathcal{O}(|E|)$
Tipp: Cut Rekonstruktion mit `unionFind` für Partitionierung oder `vector<bool>` für edge id's im cut.

```
struct edge {
  int from, to;
  ll cap;
};

vector<vector<edge>> adjlist, tmp;
vector<bool> erased;

void merge(int a, int b) {
  tmp[a].insert(tmp[a].end(), all(tmp[b]));
  tmp[b].clear();
  erased[b] = true;
  for (auto& v : tmp) {
    for (auto&e : v) {
      if (e.from == b) e.from = a;
      if (e.to == b) e.to = a;
}}}
```

```cpp
ll stoer_wagner() {
  ll res = INF;
  tmp = adjlist;
  erased.assign(sz(tmp), false);
  for (int i = 1; i < sz(tmp); i++) {
    int s = 0;
    while (erased[s]) s++;
    priority_queue<pair<ll, int>> pq;
    pq.push({0, s});
    vector<ll> con(sz(tmp));
    ll cur = 0;
    vector<pair<ll, int>> state;
    while (!pq.empty()) {
      int c = pq.top().second;
      pq.pop();
      if (con[c] < 0) continue; //already seen
      con[c] = -1;
      for (auto e : tmp[c]) {
        if (con[e.to] >= 0) {//add edge to cut
          con[e.to] += e.cap;
          pq.push({con[e.to], e.to});
          cur += e.cap;
        } else if (e.to != c) {//remove edge from cut
          cur -= e.cap;
      }}
      state.push_back({cur, c});
    }
    int t = state.back().second;
    state.pop_back();
    if (state.empty()) return 0; //graph is not connected?!
    merge(state.back().second, t);
    res = min(res, state.back().first);
  }
  return res;
}
```

2.15 Max-Flow

2.15.1 Push Relabel

`maxFlow`	gut bei sehr dicht besetzten Graphen.	$O\left(\lvert V\rvert^2 \cdot \sqrt{\lvert E\rvert}\right)$
`addEdge`	fügt eine **gerichtete** Kante ein	$O(1)$

```cpp
struct edge {
  int from, to;
  ll f, c;
};

vector<edge> edges;
vector<vector<int>> adjlist, hs;
vector<ll> ec;
vector<int> cur, H;

void addEdge(int from, int to, ll c) {
  adjlist[from].push_back(sz(edges));
  edges.push_back({from, to, 0, c});
  adjlist[to].push_back(sz(edges));
  edges.push_back({to, from, 0, 0});
}

void addFlow(int id, ll f) {
  if (ec[edges[id].to] == 0 && f > 0)
    hs[H[edges[id].to]].push_back(edges[id].to);
  edges[id].f += f;
  edges[id^1].f -= f;
  ec[edges[id].to] += f;
  ec[edges[id].from] -= f;
}

ll maxFlow(int s, int t) {
  int n = sz(adjlist);
  hs.assign(2*n, {});
  ec.assign(n, 0);
  cur.assign(n, 0);
  H.assign(n, 0);
  H[s] = n;
  ec[t] = 1;//never set t to active...
  vector<int> co(2*n);
  co[0] = n - 1;
  for (int id : adjlist[s]) addFlow(id, edges[id].c);
  for (int hi = 0;;) {
    while (hs[hi].empty()) if (!hi--) return -ec[s];
    int u = hs[hi].back();
    hs[hi].pop_back();
    while (ec[u] > 0) {
      if (cur[u] == sz(adjlist[u])) {
        H[u] = 2*n;
        for (int i = 0; i < sz(adjlist[u]); i++) {
          int id = adjlist[u][i];
          if (edges[id].c - edges[id].f > 0 &&
              H[u] > H[edges[id].to] + 1) {
            H[u] = H[edges[id].to] + 1;
            cur[u] = i;
        }}
        co[H[u]]++;
        if (!--co[hi] && hi < n) {
          for (int i = 0; i < n; i++) {
            if (hi < H[i] && H[i] < n) {
              co[H[i]]--;
              H[i] = n + 1;
        }}}
        hi = H[u];
      } else {
        auto e = edges[adjlist[u][cur[u]]];
        if (e.c - e.f > 0 && H[u] == H[e.to] + 1) {
          addFlow(adjlist[u][cur[u]], min(ec[u], e.c - e.f));
        } else {
          cur[u]++;
}}}}}
```

2.15.2 Dinic's Algorithm mit Capacity Scaling

`maxFlow`	doppelt so schnell wie Ford Fulkerson	$O\left(\lvert V\rvert^2 \cdot \lvert E\rvert\right)$
`addEdge`	fügt eine **gerichtete** Kante ein	$O(1)$

```cpp
struct edge {
  int from, to;
  ll f, c;
};

vector<edge> edges;
vector<vector<int>> adjlist;
int s, t;
vector<int> pt, dist;
ll flow, lim;
queue<int> q;

void addEdge(int from, int to, ll c) {
  adjlist[from].push_back(sz(edges));
  edges.push_back({from, to, 0, c});
  adjlist[to].push_back(sz(edges));
  edges.push_back({to, from, 0, 0});
}

bool bfs() {
  dist.assign(sz(dist), -1);
  dist[t] = sz(adjlist) + 1;
  q.push(t);
  while (!q.empty() && dist[s] < 0) {
    int cur = q.front(); q.pop();
    for (int id : adjlist[cur]) {
      int to = edges[id].to;
      if (dist[to] < 0 &&
          edges[id ^ 1].c - edges[id ^ 1].f >= lim) {
        dist[to] = dist[cur] - 1;
        q.push(to);
  }}}
  while (!q.empty()) q.pop();
  return dist[s] >= 0;
}

bool dfs(int v, ll flow) {
  if (flow == 0) return false;
  if (v == t) return true;
  for (; pt[v] < sz(adjlist[v]); pt[v]++) {
    int id = adjlist[v][pt[v]], to = edges[id].to;
    if (dist[to] == dist[v] + 1 &&
        edges[id].c - edges[id].f >= flow) {
      if (dfs(to, flow)) {
        edges[id].f += flow;
        edges[id ^ 1].f -= flow;
        return true;
  }}}
  return false;
}

ll maxFlow(int source, int target) {
  s = source;
  t = target;
  flow = 0;
  dist.resize(sz(adjlist));
  for (lim = (1LL << 62); lim >= 1;) {
    if (!bfs()) {lim /= 2; continue;}
    pt.assign(sz(adjlist), 0);
    while (dfs(s, lim)) flow += lim;
  }
  return flow;
}
```

2.16 Min-Cost-Max-Flow

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

```
constexpr ll INF = 1LL << 60; // Größer als der maximale Fluss.
struct MinCostFlow {
  struct edge {
    int to;
    ll f, cost;
  };
  vector<edge> edges;
  vector<vector<int>> adjlist;
  vector<int> pref, con;
  vector<ll> dist;

  const int s, t;
  ll maxflow, mincost;

  MinCostFlow(int n, int source, int target) :
    adjlist(n), s(source), t(target) {};

  void addedge(int u, int v, ll c, ll cost) {
    adjlist[u].push_back(sz(edges));
    edges.push_back({v, c, cost});
    adjlist[v].push_back(sz(edges));
    edges.push_back({u, 0, -cost});
  }

  bool SPFA() {
    pref.assign(sz(adjlist), - 1);
    dist.assign(sz(adjlist), INF);
    vector<bool> inqueue(sz(adjlist));
    queue<int> queue;

    dist[s] = 0; queue.push(s);
    pref[s] = s; inqueue[s] = true;

    while (!queue.empty()) {
      int cur = queue.front(); queue.pop();
      inqueue[cur] = false;
      for (int id : adjlist[cur]) {
        int to = edges[id].to;
        if (edges[id].f > 0 &&
            dist[to] > dist[cur] + edges[id].cost) {
          dist[to] = dist[cur] + edges[id].cost;
          pref[to] = cur; con[to] = id;
          if (!inqueue[to]) {
            inqueue[to] = true; queue.push(to);
    }}}}
    return pref[t] != -1;
  }

  void extend() {
    ll w = INF;
    for (int u = t; pref[u] != u; u = pref[u])
      w = min(w, edges[con[u]].f);
    maxflow += w;
    mincost += dist[t] * w;
    for (int u = t; pref[u] != u; u = pref[u]) {
      edges[con[u]].f -= w;
      edges[con[u] ^ 1].f += w;
  }}

  void mincostflow() {
    con.assign(sz(adjlist), 0);
    maxflow = mincost = 0;
    while (SPFA()) extend();
  }
};
```

2.17 Maximal Cardinatlity Bipartite Matching

`kuhn` berechnet Matching $O(|V| \cdot \min(ans^2, |E|))$

- die ersten [0..n) Knoten in `adjlist` sind die linke Seite des Graphen

```
vector<vector<int>> adjlist;
vector<int> pairs; // Der gematchte Knoten oder -1.
vector<bool> visited;

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

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

`hopcroft_karp` berechnet Matching $O\left(\sqrt{|V|} \cdot |E|\right)$

```
vector<vector<int>> adjlist;
// pairs ist der gematchte Knoten oder -1
vector<int> pairs, dist;

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

bool dfs(int u) {
  for (int v : adjlist[u]) {
    if (pairs[v] < 0 ||
       (dist[pairs[v]] > dist[u] && dfs(pairs[v]))) {
      pairs[v] = u; pairs[u] = v;
      return true;
  }}
  dist[u] = -1;
  return false;
}

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

2.18 Maximum Weight Bipartite Matching

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

```
double costs[N_LEFT][N_RIGHT];

// Es muss l<=r sein! (sonst Endlosschleife)
double match(int l, int r) {
  vector<double> lx(l), ly(r);
  //xy is matching from l->r, yx from r->l, or -1
  vector<int> xy(l, -1), yx(r, -1), augmenting(r);
  vector<bool> s(l);
  vector<pair<double, int>> slack(r);

  for (int x = 0; x < l; x++)
    lx[x] = *max_element(costs[x], costs[x] + r);
  for (int root = 0; root < l; root++) {
    augmenting.assign(r, -1);
    s.assign(l, false);
    s[root] = true;
    for (int y = 0; y < r; y++) {
      slack[y] = {lx[root] + ly[y] - costs[root][y], root};
    }
    int y = -1;
    while (true) {
      double delta = INF;
      int x = -1;
      for (int yy = 0; yy < r; yy++) {
        if (augmenting[yy] < 0) {
          if (slack[yy].first < delta) {
            delta = slack[yy].first;
            x = slack[yy].second;
            y = yy;
      }}}
      if (delta > 0) {
        for (int x = 0; x < l; x++) if (s[x]) lx[x] -= delta;
        for (int y = 0; y < r; y++) {
          if (augmenting[y] >= 0) ly[y] += delta;
          else slack[y].first -= delta;
      }}
```

```
      augmenting[y] = x;
      x = yx[y];
      if (x == -1) break;
      s[x] = true;
      for (int y = 0; y < r; y++) {
        if (augmenting[y] < 0) {
          double alt = lx[x] + ly[y] - costs[x][y];
          if (slack[y].first > alt) {
            slack[y] = {alt, x};
    }}}}
    while (y >= 0) {
      // Jede Iteration vergrößert Matching um 1
      // (können 0-Kanten sein!)
      int x = augmenting[y];
      int prec = xy[x];
      yx[y] = x;
      xy[x] = y;
      y = prec;
  }}
  // Wert des Matchings
  return accumulate(all(lx), 0.0) +
         accumulate(all(ly), 0.0);
}
```

2.19 Wert des maximalen Matchings

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

```
constexpr int MOD=1'000'000'007, I=10;
vector<vector<ll>> adjlist, mat;

int gauss(int n, ll p) {
  int rank = n;
  for (int line = 0; line < n; line++) {
    int swappee = line;
    while (swappee < n && mat[swappee][line] == 0) swappee++;
    if (swappee == n) {rank--; continue;}
    swap(mat[line], mat[swappee]);
    ll factor = powMod(mat[line][line], p - 2, p);
    for (int i = 0; i < n; i++) {
      mat[line][i] *= factor;
      mat[line][i] %= p;
    }
    for (int i = 0; i < n; i++) {
      if (i == line) continue;
      ll diff = mat[i][line];
      for (int j = 0; j < n; j++) {
        mat[i][j] -= (diff * mat[line][j]) % p;
        mat[i][j] %= p;
        if (mat[i][j] < 0) mat[i][j] += p;
  }}}
  return rank;
}

int max_matching() {
  int ans = 0;
  mat.assign(sz(adjlist), vector<ll>(sz(adjlist)));
  for (int _ = 0; _ < I; _++) {
    for (int i = 0; i < sz(adjlist); i++) {
      mat[i].assign(sz(adjlist), 0);
      for (int j : adjlist[i]) {
        if (j < i) {
          mat[i][j] = rand() % (MOD - 1) + 1;
          mat[j][i] = MOD - mat[i][j];
    }}}
    ans = max(ans, gauss(sz(adjlist), MOD)/2);
  }
  return ans;
}
```

2.20 Allgemeines maximales Matching

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

```
struct GM {
  vector<vector<int>> adjlist;
  // pairs ist der gematchte knoten oder n
  vector<int> pairs, first, que;
  vector<pair<int, int>> label;
  int head, tail;

  GM(int n) : adjlist(n), pairs(n + 1, n), first(n + 1, n),
              que(n), label(n + 1, {-1, -1}) {}

  void rematch(int v, int w) {
    int t = pairs[v]; pairs[v] = w;
    if (pairs[t] != v) return;
    if (label[v].second == -1) {
      pairs[t] = label[v].first;
      rematch(pairs[t], t);
    } else {
      auto [x, y] = label[v];
      rematch(x, y);
      rematch(y, x);
  }}

  int findFirst(int u) {
    return label[first[u]].first < 0 ? first[u]
         : first[u] = findFirst(first[u]);
  }

  void relabel(int x, int y) {
    int r = findFirst(x);
    int s = findFirst(y);
    if (r == s) return;
    auto h = label[r] = label[s] = {~x, y};
    int join;
    while (true) {
      if (s != sz(adjlist)) swap(r, s);
      r = findFirst(label[pairs[r]].first);
      if (label[r] == h) {
        join = r;
        break;
      } else {
        label[r] = h;
    }}
    for (int v : {first[x], first[y]}) {
      for (; v != join; v = first[label[pairs[v]].first]) {
        label[v] = {x, y};
        first[v] = join;
        que[tail++] = v;
  }}}

  bool augment(int u) {
    label[u] = {sz(adjlist), -1};
    first[u] = sz(adjlist);
    head = tail = 0;
    for (que[tail++] = u; head < tail;) {
      int x = que[head++];
      for (int y : adjlist[x]) {
        if (pairs[y] == sz(adjlist) && y != u) {
          pairs[y] = x;
          rematch(x, y);
          return true;
        } else if (label[y].first >= 0) {
          relabel(x, y);
        } else if (label[pairs[y]].first == -1) {
          label[pairs[y]].first = x;
          first[pairs[y]] = y;
          que[tail++] = pairs[y];
    }}}
    return false;
  }

  int match() {
    int matching = head = tail = 0;
    for (int u = 0; u < sz(adjlist); u++) {
      if (pairs[u] < sz(adjlist) || !augment(u)) continue;
      matching++;
      for (int i = 0; i < tail; i++)
        label[que[i]] = label[pairs[que[i]]] = {-1, -1};
      label[sz(adjlist)] = {-1, -1};
    }
    return matching;
  }
};
```

2.21 Cycle Counting

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

count zählt Zykel $O(2^{|base|})$

- jeder Zyklus ist das xor von einträgen in base.

```
constexpr int maxEdges = 128;
using cycle = bitset<maxEdges>;
struct cylces {
  vector<vector<pair<int, int>>> adj;
  vector<bool> seen;
  vector<cycle> paths, base;
  vector<pair<int, int>> edges;

  cylces(int n) : adj(n), seen(n), paths(n) {}

  void addEdge(int a, int b) {
    adj[a].push_back({b, sz(edges)});
    adj[b].push_back({a, sz(edges)});
    edges.push_back({a, b});
  }

  void addBase(cycle cur) {
    for (cycle o : base) {
      o ^= cur;
      if (o._Find_first() > cur._Find_first()) cur = o;
```

```
    }
    if (cur.any()) base.push_back(cur);
  }
  void findBase(int c = 0, int p = -1, cycle cur = {}) {
    if (adj.empty()) return;
    if (seen[c]) {
      addBase(cur ^ paths[c]);
    } else {
      seen[c] = true;
      paths[c] = cur;
      for (auto [to, id] : adj[c]) {
        if (to == p) continue;
        cur[id].flip();
        findBase(to, c, cur);
        cur[id].flip();
  }}}
  //cycle must be constrcuted from base
  bool isCycle(cycle cur) {
    if (cur.none()) return false;
    init(sz(adj)); // union find
    for (int i = 0; i < sz(edges); i++) {
      if (cur[i]) {
        cur[i] = false;
        if (findSet(edges[i].first) ==
            findSet(edges[i].second)) break;
        unionSets(edges[i].first, edges[i].second);
    }}
    return cur.none();
  };
  int count() {
    findBase();
    int res = 0;
    for (int i = 1; i < (1 << sz(base)); i++) {
      cycle cur;
      for (int j = 0; j < sz(base); j++) {
        if (((i >> j) & 1) != 0) cur ^= base[j];
      if (isCycle(cur)) res++;
    }
    return res;
  }
};
```

2.22 Eulertouren

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

```
vector<vector<int>> idx;
vector<int> to, validIdx, cycle;
vector<bool> used;
void addEdge(int a, int b) {
  idx[a].push_back(sz(to));
  to.push_back(b);
  used.push_back(false);
  idx[b].push_back(sz(to)); // für ungerichtet
  to.push_back(a);
  used.push_back(false);
}
void euler(int n) { // init idx und validIdx
  for (;validIdx[n] < sz(idx[n]); validIdx[n]++) {
    if (!used[idx[n][validIdx[n]]]) {
      int nn = to[idx[n][validIdx[n]]];
      used[idx[n][validIdx[n]]] = true;
      used[idx[n][validIdx[n]] ^ 1] = true; // für ungerichtet
      euler(nn);
  }}
  cycle.push_back(n); // Zyklus in umgekehrter Reihenfolge.
}
```

- 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>> adjlist` leichter
- **Wichtig:** Algorithmus schlägt nicht fehl, falls kein Eulerzyklus existiert. Die Existenz muss separat geprüft werden.

2.23 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	$O(\log(n))$

```
struct connect {
  int n;
  vector<pair<int, int>> edges;
  LCT lct; // min LCT no updates required
  connect(int n, int m) : n(n), edges(m), lct(n+m) {}
  bool connected(int a, int b) {
    return lct.connected(&lct.nodes[a], &lct.nodes[b]);
  }
  void addEdge(int a, int b, int id) {
    lct.nodes[id + n] = LCT::Node(id + n, id + n);
    edges[id] = {a, b};
    if (connected(a, b)) {
      int old = lct.query(&lct.nodes[a], &lct.nodes[b]);
      if (old < id) eraseEdge(old);
    }
    if (!connected(a, b)) {
      lct.link(&lct.nodes[a], &lct.nodes[id + n]);
      lct.link(&lct.nodes[b], &lct.nodes[id + n]);
  }}
  void eraseEdge(ll id) {
    if (connected(edges[id].first, edges[id].second) &&
      lct.query(&lct.nodes[edges[id].first],
                &lct.nodes[edges[id].second]) == id) {
      lct.cut(&lct.nodes[edges[id].first], &lct.nodes[id + n]);
      lct.cut(&lct.nodes[edges[id].second], &lct.nodes[id + n]);
  }}
};
```

3 Geometrie

3.1 Closest Pair

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

```
bool compY(pt a, pt b) {
  return (imag(a) == imag(b)) ? real(a) < real(b)
                              : imag(a) < imag(b);
}
bool compX(pt a, pt b) {
  return (real(a) == real(b)) ? imag(a) < imag(b)
                              : real(a) < real(b);
}
double shortestDist(vector<pt>& pts) { // sz(pts) > 1
  set<pt, bool(*)(pt, pt)> status(compY);
  sort(all(pts), compX);
  double opt = 1.0/0.0, sqrtOpt = 1.0/0.0;
  auto left = pts.begin(), right = pts.begin();
  status.insert(*right); right++;
  while (right != pts.end()) {
    if (left != right &&
        abs(real(*left - *right)) >= sqrtOpt) {
      status.erase(*left);
      left++;
    } else {
      auto lower = status.lower_bound({-1.0/0.0, //-INF
                                 imag(*right) - sqrtOpt});
      auto upper = status.upper_bound({-1.0/0.0, //-INF
                                 imag(*right) + sqrtOpt});
      for (;lower != upper; lower++) {
        double cand = norm(*right - *lower);
        if (cand < opt) {
          opt = cand;
          sqrtOpt = sqrt(opt);
      }}
      status.insert(*right);
      right++;
  }}
  return sqrtOpt;
}
```

3.2 Rotating calipers

`antipodalPoints` berechnet antipodale Punkte $O(n)$

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

```
vector<pair<int, int>> antipodalPoints(vector<pt>& h) {
  if (sz(h) < 2) return {};
  vector<pair<int, int>> result;
  for (int i = 0, j = 1; i < j; i++) {
    while (true) {
      result.push_back({i, j});
      if (cross(h[(i + 1) % sz(h)] - h[i],
                h[(j + 1) % sz(h)] - h[j]) <= 0) break;
      j = (j + 1) % sz(h);
  }}
  return result;
}
```

3.3 Konvexe Hülle

`convexHull` berechnet Konvexehülle $O(n \cdot \log(n))$

- Konvexehülle gegen den Uhrzeigersinn Sortiert
- nur Eckpunkte enthalten(für alle Punkte = im CCW Test entfernen)
- Erster und Letzter Punkt sind identisch

```
vector<pt> convexHull(vector<pt> pts){
  sort(all(pts), [](const pt& a, const pt& b){
    return real(a) == real(b) ? imag(a) < imag(b)
                              : real(a) < real(b);
  });
  pts.erase(unique(all(pts)), pts.end());
  int k = 0;
  vector<pt> h(2 * sz(pts));
  for (int i = 0; i < sz(pts); i++) {// Untere Hülle.
    while (k > 1 && cross(h[k-2], h[k-1], pts[i]) <= 0) k--;
    h[k++] = pts[i];
  }
  for (int i = sz(pts)-2, t = k; i >= 0; i--) {// Obere Hülle.
    while (k > t && cross(h[k-2], h[k-1], pts[i]) <= 0) k--;
    h[k++] = pts[i];
  }
  h.resize(k);
  return h;
}
```

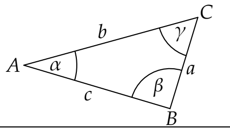
3.4 Formeln – `std::complex`

```
// Komplexe Zahlen als Punkte. Wenn immer möglich complex<ll>
// verwenden. Funktionen wie abs() geben dann aber ll zurück.
using pt = complex<double>;

constexpr double PIU = acos(-1.0l); // PIL < PI < PIU
constexpr double PIL = PIU-2e-19l;

// Winkel zwischen Punkt und x-Achse in [-PI, PI].
double angle(pt a) {return arg(a);}

// rotiert Punkt im Uhrzeigersinn um den Ursprung.
pt rotate(pt a, double theta) {return a * polar(1.0, theta);}

// Skalarprodukt.
double dot(pt a, pt b) {return real(conj(a) * b);}

// abs()^2.(pre c++20)
double norm(pt a) {return dot(a, a);}

// Kreuzprodukt, 0, falls kollinear.
double cross(pt a, pt b) {return imag(conj(a) * b);}
double cross(pt p, pt a, pt b) {return cross(a - p, b - p);}

//  1 => c links von a->b
//  0 => a, b und c kolliniear
// -1 => c rechts von a->b
int orientation(pt a, pt b, pt c) {
  double orien = cross(b - a, c - a);
  return (orien > EPS) - (orien < -EPS);
}

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

// identifiziert winkel zwischen Vektoren u und v
pt uniqueAngle(pt u, pt v) {
  pt tmp = v * conj(u);
  ll g = abs(gcd(real(tmp), imag(tmp)));
  return tmp / g;
}
```

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

// Berechnet die Schnittpunkte der Strecken p0-p1 und p2-p3.
// Enthält entweder keinen Punkt, den einzigen Schnittpunkt
// oder die Endpunkte der Schnittstrecke.
vector<pt> lineSegmentIntersection(pt p0, pt p1, pt p2, pt p3) {
  double a = cross(p1 - p0, p3 - p2);
  double b = cross(p2 - p0, p3 - p2);
  double c = cross(p1 - p0, p0 - p2);
  if (a < 0) {a = -a; b = -b; c = -c;}
  if (b < -EPS || a-b < -EPS ||
      c < -EPS || a-c < -EPS) return {};
  if (a > EPS) return {p0 + b/a*(p1 - p0)};
  vector<pt> result;
  auto insertUnique = [&](pt p) {
    for (auto q: result) if (abs(p - q) < EPS) return;
    result.push_back(p);
  };
  if (dot(p2-p0, p3-p0) < EPS) insertUnique(p0);
  if (dot(p2-p1, p3-p1) < EPS) insertUnique(p1);
  if (dot(p0-p2, p1-p2) < EPS) insertUnique(p2);
  if (dot(p0-p3, p1-p3) < EPS) insertUnique(p3);
  return result;
}

// Entfernung von Punkt p zur Gearden durch a-b. 2d und 3d
double distToLine(pt a, pt b, pt p) {
  return abs(cross(p - a, b - a)) / abs(b - a);
}

// Projektiert p auf die Gerade a-b
pt projectToLine(pt a, pt b, pt p) {
  return a + (b - a) * dot(p - a, b - a) / norm(b - a);
}

// Liegt p auf der Geraden a-b? 2d und 3d
bool pointOnLine(pt a, pt b, pt p) {
  return cross(a, b, p) == 0;
}

// Test auf Linienschnitt zwischen a-b und c-d.
bool lineIntersection(pt a, pt b, pt c, pt d) {
  return abs(cross(a - b, c - d)) < EPS;
}

// Berechnet den Schnittpunkt der Graden p0-p1 und p2-p3.
// die Graden dürfen nicht parallel sein!
pt lineIntersection(pt p0, pt p1, pt p2, pt p3) {
  double a = cross(p1 - p0, p3 - p2);
  double b = cross(p2 - p0, p3 - p2);
  return {p0 + b/a*(p1 - p0)};
}

// Liegt p auf der Strecke a-b?
bool pointOnLineSegment(pt a, pt b, pt p) {
  if (cross(a, b, p) != 0) return false;
  double dist = norm(a - b);
  return norm(a - p) <= dist && norm(b - p) <= dist;
}

// Entfernung von Punkt p zur Strecke a-b.
double distToSegment(pt a, pt b, pt p) {
  if (a == b) return abs(p - a);
  if (dot(p - a, b - a) <= 0) return abs(p - a);
  if (dot(p - b, b - a) >= 0) return abs(p - b);
  return distToLine(a, b, p);
}

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

// sortiert alle Punkte pts auf einer Linie entsprechend dir
void sortLine(pt dir, vector<pt>& pts) { // (2d und 3d)
  sort(all(pts), [&](pt a, pt b){
    return dot(dir, a) < dot(dir, b);
  });
}
```

Generell:

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



$\beta = 90°$:

- $\sin(\alpha) = \frac{a}{b}$
- $\cos(\alpha) = \frac{c}{b}$
- $\tan(\alpha) = \frac{a}{c}$

```
// Mittelpunkt des Dreiecks abc.
pt centroid(pt a, pt b, pt c) {return (a + b + c) / 3.0;}

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

// Flächeninhalt eines Dreiecks bei bekannten Seitenlängen.
double area(double a, double b, double c) {
  double s = (a + b + c) / 2.0;
  return sqrt(s * (s-a) * (s-b) * (s-c));
}

// Zentrum des größten Kreises im Dreiecke
pt inCenter(pt a, pt b, pt c) {
  double x = abs(a-b), y = abs(b-c), z = abs(a-c);
  return (y*a + z*b + x*c) / (x+y+z);
}
```

```
17 // Zentrum des Kreises durch alle Eckpunkte
18 pt outCenter(pt a, pt b, pt c) {
19   double d = 2.0 * (real(a) * imag(b-c) +
20                     real(b) * imag(c-a) +
21                     real(c) * imag(a-b));
22   return (a*conj(a)*conj(b-c) +
23           b*conj(b)*conj(c-a) +
24           c*conj(c)*conj(a-b)) / d;
25 }
26 // Sind die Dreiecke a1, b1, c1, and a2, b2, c2 ähnlich?
27 // Erste Zeile testet Ähnlichkeit mit gleicher Orientierung,
28 // zweite Zeile testet Ähnlichkeit mit verschiedener Orientierung
29 bool similar (pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
30   return ((b2-a2) * (c1-a1) == (b1-a1) * (c2-a2) ||
31           (b2-a2) * (conj(c1)-conj(a1)) == (conj(b1)-conj(a1))
32         * (c2-a2)
33   );
34 }
```

```
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   double 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 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 += orientation(p, poly[i], poly[i + 1]);
21   }}
22   return res;
23 }
24 // Testet, ob ein Punkt im Polygon liegt (beliebige Polygone).
25 // Ändere Zeile 32 falls rand zählt, poly[0] == poly.back()
26 bool inside(pt p, const vector<pt>& poly) {
27   bool in = false;
28   for (int i = 0; i + 1 < sz(poly); i++) {
29     pt a = poly[i], b = poly[i + 1];
30     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 // Change line 45 and 51 >= if border counts as inside
40 bool inside(pt p, const vector<pt>& hull) {
41   int l = 0, r = sz(hull) - 1;
42   if (cross(hull[0], hull[r], p) > 0) return false;
43   while (l + 1 < r) {
44     int m = (l + r) / 2;
45     if (cross(hull[0], hull[m], p) >= 0) l = m;
46     else r = m;
47   }
48   return cross(hull[l], hull[r], p) > 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, const 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 = 0.0;
80   //bool intersect = true;
81   for (ll i = 0; i + 1 < sz(p); i++) {
82     //intersect &= cross(p[i], p[i+1] - p[i]) <= 0;
83     res = max(res, cross(p[i], p[i+1]-p[i]) / abs(p[i+1]-p[i]));
84   }
85   return 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  return r;
107 }
108 // convex hulls without duplicates, hull[0] == hull.back() and
109 // hull[0] must be a convex point (with angle < pi)
110 // {} if no intersection
111 // {x} if corner is only intersection
112 // {a, b} segments (a,a+1) and (b,b+1) intersected (if only the
113 // border is intersected corners a and b are the start and end)
114 vector<int> intersect(const vector<pt>& hull, pt a, pt b) {
115   int endA = extremal(hull, (a-b) * pt(0, 1));
116   int endB = extremal(hull, (b-a) * pt(0, 1));
117   // cross == 0 => line only intersects border
118   if (cross(hull[endA], a, b) > 0 ||
119       cross(hull[endB], a, b) < 0) return {};
120   int n = sz(hull) - 1;
121   vector<int> res;
122   for (auto _ : {0, 1}) {
123     int l = endA, r = endB;
124     if (r < l) r += n;
125     while (l + 1 < r) {
126       int m = (l + r) / 2;
127       if (cross(hull[m % n], a, b) <= 0 &&
128           cross(hull[m % n], a, b) != hull(poly[endB], a, b))
129         l = m;
130       else r = m;
131     }
132     if (cross(hull[r % n], a, b) == 0) l++;
133     res.push_back(l % n);
134     swap(endA, endB);
135     swap(a, b);
136   }
137   if (res[0] == res[1]) res.pop_back();
138   return res;
139 }
```

```
1 bool left(pt p) {return real(p) < 0 ||
2                        (real(p) == 0 && imag(p) < 0);}
3 void sortAround(pt p, vector<pt>& ps) {
4   sort(all(ps), [&](const pt& a, const pt& b){
5     if (left(a - p) != left(b - p))
6       return left(a - p) > left(b - p);
7     return cross(p, a, b) > 0;
8   });
9 }
```

```cpp
// berechnet die Schnittpunkte von zwei kreisen
// (Kreise dürfen nicht gleich sein!)
vector<pt> circleIntersection(pt c1, double r1,
                              pt c2, double r2) {
  double d = abs(c1 - c2);
  if (d < abs(r1 - r2) || d > abs(r1 + r2)) return {};
  double a = (r1 * r1 - r2 * r2 + d * d) / (2 * d);
  pt p = (c2 - c1) * a / d + c1;
  if (d == abs(r1 - r2) || d == abs(r1 + r2)) return {p};
  double h = sqrt(r1 * r1 - a * a);
  return {p + pt{0, 1} * (c2 - c1) * h / d,
          p - pt{0, 1} * (c2 - c1) * h / d};
}
// berechnet die Schnittpunkte zwischen
// einem Kreis(Kugel) und einer Grade 2d und 3d
vector<pt> circleRayIntersection(pt center, double r,
                                 pt orig, pt dir) {
  vector<pt> result;
  double a = dot(dir, dir);
  double b = 2 * dot(dir, orig - center);
  double c = dot(orig - center, orig - center) - r * r;
  double discr = b * b - 4 * a * c;
  if (discr >= 0) {
    //t in [0, 1] => schnitt mit segment [orig, orig + dir]
    double t1 = -(b + sqrt(discr)) / (2 * a);
    double t2 = -(b - sqrt(discr)) / (2 * a);
    if (t1 >= 0) result.push_back(t1 * dir + orig);
    if (t2 >= 0 && abs(t1 - t2) > EPS) {
      result.push_back(t2 * dir + orig);
  }}
  return result;
}
```

3.5 Formeln - 3D

```cpp
// Skalarprodukt
double operator|(pt3 a, pt3 b) {
  return a.x * b.x + a.y*b.y + a.z*b.z;
}
double dot(pt3 a, pt3 b) {return a|b;}
// Kreuzprodukt
pt3 operator*(pt3 a, pt3 b) {return {a.y*b.z - a.z*b.y,
                                     a.z*b.x - a.x*b.z,
                                     a.x*b.y - a.y*b.x};}
pt3 cross(pt3 a, pt3 b) {return a*b;}
// Länge von a
double abs(pt3 a) {return sqrt(dot(a, a));}
double abs(pt3 a, pt3 b) {return abs(b - a);}
// Mixedprodukt
double mixed(pt3 a, pt3 b, pt3 c) {return a*b|c;};
// orientierung von p zu der Ebene durch a, b, c
// -1 => gegen den Uhrzeigersinn,
//  0 => kolliniear,
//  1 => im Uhrzeigersinn.
int orientation(pt3 a, pt3 b, pt3 c, pt3 p) {
  double orien = mixed(b - a, c - a, p - a);
  return (orien > EPS) - (orien < -EPS);
}
// Entfernung von Punkt p zur Ebene a,b,c.
double distToPlane(pt3 a, pt3 b, pt3 c, pt3 p) {
  pt3 n = cross(b-a, c-a);
  return (abs(dot(n, p)) - dot(n, a)) / abs(n);
}
// Liegt p in der Ebene a,b,c?
bool pointOnPlane(pt3 a, pt3 b, pt3 c, pt3 p) {
  return orientation(a, b, c, p) == 0;
}
// Schnittpunkt von der Grade a-b und der Ebene c,d,e
// die Grade darf nicht parallel zu der Ebene sein!
pt3 linePlaneIntersection(pt3 a, pt3 b, pt3 c, pt3 d, pt3 e) {
  pt3 n = cross(d-c, e-c);
  pt3 d = b - a;
  return a - d * (dot(n, a) - dot(n, c)) / dot(n, d);
}
// Abstand zwischen der Grade a-b und c-d
double lineLineDist(pt3 a, pt3 b, pt3 c, pt3 d) {
  pt3 n = cross(b - a, d - c);
  if (abs(n) < EPS) return distToLine(a, b, c);
  return abs(dot(a - c, n)) / abs(n);
}
```

4 Mathe

4.1 Longest Increasing Subsequence

- lower_bound ⇒ streng monoton
- upper_bound ⇒ monoton

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

4.2 Zykel Erkennung

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

```cpp
void cycleDetection(ll x0, function<ll(ll)> f) {
  ll a = x0, b = f(x0), length = 1;
  for (ll power = 1; a != b; b = f(b), length++) {
    if (power == length) {
      power *= 2;
      length = 0;
      a = b;
  }}
  ll start = 0;
  a = x0; b = x0;
  for (ll i = 0; i < length; i++) b = f(b);
  while (a != b) {
    a = f(a);
    b = f(b);
    start++;
}}
```

4.3 Permutationen

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

```cpp
vector<ll> kthperm(ll k, ll n) {
  Tree<ll> t;
  vector<ll> res(n);
  for (ll i = 1; i <= n; k /= i, i++) {
    t.insert(i - 1);
    res[n - i] = k % i;
  }
  for (ll& x : res) {
    auto it = t.find_by_order(x);
    x = *it;
    t.erase(it);
  }
  return res;
}
```

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

```cpp
ll permIndex(vector<ll> v) {
  Tree<ll> t;
  reverse(all(v));
  for (ll& x : v) {
    t.insert(x);
    x = t.order_of_key(x);
  }
  ll res = 0;
  for (ll i = sz(v); i > 0; i--) {
    res *= i;
    res += v[i - 1];
  }
  return res;
}
```

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

`mulMod` berechnet $a \cdot b \bmod n$ $\mathcal{O}(\log(b))$

```
ll mulMod(ll a, ll b, ll n) {
  ll res = 0;
  while (b > 0) {
    if (b & 1) res = (a + res) % n;
    a = (a * 2) % n;
    b /= 2;
  }
  return res;
}
```

`powMod` berechnet $a^b \bmod n$ $\mathcal{O}(\log(b))$

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

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

4.5 ggT, kgV, erweiterter euklidischer Algorithmus

$\mathcal{O}(\log(a)+\log(b))$

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

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

4.6 Multiplikatives Inverses von n in $\mathbb{Z}/p\mathbb{Z}$

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

Falls $\mathrm{ggT}(n,p)=1$:

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

Sonst $\mathrm{ggT}(n,p)>1$: Es existiert kein x^{-1}.

```
ll multInv(ll n, ll p) {
  ll x, y;
  extendedEuclid(n, p, x, y); // Implementierung von oben.
  return ((x % p) + p) % p;
}
```

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

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

Pell-Gleichungen Sei $(\overline{x},\overline{y})$ 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:

$$x_1 := \tilde{x}, \qquad y_1 := \tilde{y}$$
$$x_{k+1} := \overline{x}x_k + n\overline{y}y_k, \qquad y_{k+1} := \overline{x}y_k + \overline{y}x_k$$

4.7 Lineare Kongruenz

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

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

4.8 Chinesischer Restsatz

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

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

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

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

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

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

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

4.9 Primzahltest & Faktorisierung

`isPrime` prüft ob Zahl prim ist $\mathcal{O}(\log(n)^2)$

```
constexpr ll bases32[] = {2, 7, 61};
constexpr ll bases64[] = {2, 325, 9375, 28178, 450775,
                          9780504, 1795265022};
bool isPrime(ll n) {
  if (n < 2 || n % 2 == 0) return n == 2;
  ll d = n - 1, j = 0;
  while (d % 2 == 0) d /= 2, j++;
  for (ll a : bases64) {
    if (a % n == 0) continue;
    ll v = powMod(a, d, n); //with mulmod or int128
    if (v == 1 || v == n - 1) continue;
    for (ll i = 1; i <= j; i++) {
      v = (v * v) % n; //mulmod or int128
      if (v == n - 1 || v <= 1) break;
    }
    if (v != n - 1) return false;
  }
  return true;
}
```

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

```
using lll = __int128;
ll rho(ll n) { // Findet Faktor < n, nicht unbedingt prim.
  if (n % 2 == 0) return 2;
  ll x = 0, y = 0, prd = 2;
  auto f = [n](lll x){return (x * x) % n + 1;};
  for (ll t = 30, i = n/2 + 7; t % 40 || gcd(prd, n) == 1; t++) {
    if (x == y) x = ++i, y = f(x);
    if (ll q = (lll)prd * abs(x-y) % n; q) prd = q;
    x = f(x); y = f(f(y));
  }
  return gcd(prd, n);
}

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

4.10 Teiler

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

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

4.11 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 $\mathcal{O}(\log(\varphi(n))\cdot\log(n))$
`findPrimitive` findet Primitivwurzel (oder -1) $\mathcal{O}(|ans|\cdot\log(\varphi(n))\cdot\log(n))$

```cpp
bool isPrimitive(ll g, ll n, ll phi, map<ll, int> phiFacs) {
  if (g == 1) return n == 2;
  for (auto [f, _] : phiFacs)
    if (powMod(g, phi / f, n) == 1) return false;
  return true;
}
bool isPrimitive(ll g, ll n) {
  ll phin = phi(n); //isPrime(n) => phi(n) = n - 1
  map<ll, int> phiFacs;
  factor(phin, phiFacs);
  return isPrimitive(g, n, phin, phiFacs);
}
ll findPrimitive(ll n) {
  ll phin = phi(n); //isPrime(n) => phi(n) = n - 1
  map<ll, int> phiFacs;
  factor(phin, phiFacs);
  //auch zufällige Reihenfolge möglich!
  for (ll res = 1; res < n; res++)
    if (isPrimitive(res, n, phin, phiFacs)) return res;
  return -1;
}
```

4.12 Diskreter Logarithmus

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

```cpp
ll dlog(ll a, ll b, ll m) {
  ll bound = sqrtl(m) + 1; //memory usage bound
  map<ll, ll> vals;
  for (ll i = 0, e = 1; i < bound; i++, e = (e * a) % m) {
    vals[e] = i;
  }
  ll fact = powMod(a, m - bound - 1, m);
  for (ll i = 0; i < m; i += bound, b = (b * fact) % m) {
    if (vals.count(b)) {
      return i + vals[b];
  }}
  return -1;
}
```

4.13 Diskrete n-te Wurzel

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

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

```cpp
ll root(ll a, ll b, ll m) {
  ll g = findPrimitive(m);
  ll c = dlog(powMod(g, a, m), b, m); //diskreter logarithmus
  return c < 0 ? -1 : powMod(g, c, m);
}
```

4.14 Linearessieb und Multiplikative Funktionen

Eine (zahlentheoretische) Funktion f heißt multiplikativ wenn $f(1)=1$ und $f(a\cdot b)=f(a)\cdot f(b)$, falls $\mathrm{ggT}(a,b)=1$.
$\Rightarrow$ Es ist ausreichend $f(p^k)$ für alle primen p und alle k zu kennen.

`sieve` berechnet Primzahlen und co. $\mathcal{O}(N)$
`sieved` Wert der endsprechenden Multiplikativen Funktion $\mathcal{O}(1)$
`naive` Wert der endsprechenden Multiplikativen Funktion $\mathcal{O}(\sqrt{n})$

Wichtig: Sieb rechts ist schneller für `isPrime` oder `primes`!

```cpp
constexpr ll N = 10'000'000;
ll smallest[N], power[N], sieved[N];
vector<ll> primes;
//wird aufgerufen mit (p^k, p, k) für prime p
ll mu(ll pk, ll p, ll k) {return -(k == 1);}
ll phi(ll pk, ll p, ll k) {return pk - pk / p;}
ll div(ll pk, ll p, ll k) {return k+1;}
ll divSum(ll pk, ll p, ll k) {return (pk*p+1) / (p - 1);}
ll square(ll pk, ll p, ll k) {return k % 2 ? pk / p : pk;}
ll squareFree(ll pk, ll p, ll k) {return k % 2 ? pk : 1;}
void sieve() { // O(N)
  smallest[1] = power[1] = sieved[1] = 1;
  for (ll i = 2; i < N; i++) {
    if (smallest[i] == 0) {
      primes.push_back(i);
      for (ll pk = i, k = 1; pk < N; pk *= i, k++) {
        smallest[pk] = i;
        power[pk] = pk;
        sieved[pk] = mu(pk, i, k); // Aufruf ändern!
    }}
    for (ll j = 0;
         i * primes[j] < N && primes[j] < smallest[i]; j++) {
      ll k = i * primes[j];
      smallest[k] = power[k] = primes[j];
      sieved[k] = sieved[i] * sieved[primes[j]];
    }
    if (i * smallest[i] < N && power[i] != i) {
      ll k = i * smallest[i];
      smallest[k] = smallest[i];
      power[k] = power[i] * smallest[i];
      sieved[k] = sieved[power[k]] * sieved[k / power[k]];
}}}
ll naive(ll n) { // O(sqrt(n))
  ll res = 1;
  for (ll p = 2; p * p <= n; p++) {
    if (n % p == 0) {
      ll pk = 1;
      ll k = 0;
      do {
        n /= p;
        pk *= p;
        k++;
      } while (n % p == 0);
      res *= mu(pk, p, k); // Aufruf ändern!
  }}
  return res;
}
```

Möbius Funtkion:

- $\mu(n)=+1$, 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 φ-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: $\gcd(a,c)=1 \Leftrightarrow a^b\equiv a^{b \bmod \varphi(c)} \pmod c$. Falls m prim ist, liefert das den **kleinen Satz von Fermat**: $a^m\equiv a \pmod m$

4.15 Primzahlsieb von Eratosthenes

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

`primeSieve` berechnet Primzahlen und Anzahl $\mathcal{O}(N\cdot\log(\log(N)))$
`isPrime` prüft ob Zahl prim ist $\mathcal{O}(1)$

```cpp
constexpr ll N = 100'000'000;
bitset<N / 2> isNotPrime;
vector<ll> primes = {2};
bool isPrime(ll x) {
  if (x < 2 || x % 2 == 0) return x == 2;
  else return !isNotPrime[x / 2];
}
void primeSieve() {
  for (ll i = 3; i < N; i += 2) {// i * i < N reicht für isPrime
    if (!isNotPrime[i / 2]) {
      primes.push_back(i); // optional
      for (ll j = i * i; j < N; j+= 2 * i) {
        isNotPrime[j / 2] = 1;
}}}}
```

4.16 Möbius-Inversion

- Seien $f,g:\mathbb{N}\to\mathbb{N}$ 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 $A=a_1,\ldots,a_n$ von Zahlen, $1\leq a_i\leq N$. Zähle die Anzahl der *coprime subsequences*.
Lösung: Für jedes x, sei $cnt[x]$ die Anzahl der Vielfachen von x in A. Es gibt $2^{[x]}-1$ nicht leere Subsequences in A, die nur Vielfache von x enthalten. Die Anzahl der Subsequences mit ggT$=1$ ist gegeben durch $\sum_{i=1}^{N}\mu(i)\cdot(2^{cnt[i]}-1)$.

4.17 Numerisch Extremstelle bestimmen

```cpp
ld gss(ld l, ld r, function<ld(ld)> f) {
  ld inv = (sqrt(5.0l) - 1) / 2;
  ld x1 = r - inv*(r-l), x2 = l + inv*(r-l);
  ld f1 = f(x1), f2 = f(x2);
  for (int i = 0; i < 200; i++) {
    if (f1 < f2) { //change to > to find maximum
      u = x2; x2 = x1; f2 = f1;
      x1 = r - inv*(r-l); f1 = f(x1);
    } else {
      l = x1; x1 = x2; f1 = f2;
      x2 = l + inv*(r-l); f2 = f(x2);
    }
  }
  return l;
}
```

4.18 Numerisch Integrieren, Simpsonregel

```
double f(double x) {return x;}
double simps(double a, double b) {
  return (f(a) + 4.0 * f((a + b) / 2.0) + f(b)) * (b - a) / 6.0;
}
double integrate(double a, double b) {
  double m = (a + b) / 2.0;
  double l = simps(a, m), r = simps(m, b), tot = simps(a, b);
  if (abs(l + r - tot) < EPS) return tot;
  return integrate(a, m) + integrate(m, b);
}
```

4.19 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 $p \geq 2^{\text{bits}}$

```
/*constexpr ll mod = 998244353; NTT only
constexpr ll root = 3;*/
using cplx = complex<double>;
//void fft(vector<ll> &a, bool inverse = 0) { NTT, xor, or, and
void fft(vector<cplx>& a, bool inverse = 0) {
  int n = a.size();
  for (int i = 0, j = 1; j < n - 1; ++j) {
    for (int k = n >> 1; k > (i ^= k); k >>= 1);
    if (j < i) swap(a[i], a[j]);
  }
  for (int s = 1; s < n; s *= 2) {
    /*ll ws = powMod(root, (mod - 1) / s >> 1, mod); NTT only
    if (inverse) ws = powMod(ws, mod - 2, mod);*/
    double angle = PI / s * (inverse ? -1 : 1);
    cplx ws(cos(angle), sin(angle));
    for (int j = 0; j < n; j+= 2 * s) {
      //ll w = 1; NTT only
      cplx w = 1;
      for (int k = 0; k < s; k++) {
        /*ll u = a[j + k], t = a[j + s + k] * w; NTT only
        t %= mod;
        a[j + k] = (u + t) % mod;
        a[j + s + k] = (u - t + mod) % mod;
        w = (w * ws) % mod;*/
        /*ll u = a[j + k], t = a[j + s + k]; xor only
        a[j + k] = u + t;
        a[j + s + k] = u - t;*/
        /*if (!inverse) { or only
          a[j + k] = u + t;
          a[j + s + k] = u;
        } else {
          a[j + k] = t;
          a[j + s + k] = u - t;
        }*/
        /*if (!inverse) { and only
          a[j + k] = t;
          a[j + s + k] = u + t;
        } else {
          a[j + k] = t - u;
          a[j + s + k] = u;
        }*/
        cplx u = a[j + k], t = a[j + s + k] * w;
        a[j + k] = u + t;
        a[j + s + k] = u - t;
        if (inverse) a[j + k] /= 2, a[j + s + k] /= 2;
        w *= ws;
  }}}
  /*if (inverse) { NTT only
    ll div = powMod(n, mod - 2, mod);
    for (ll i = 0; i < n; i++) {
      a[i] = (a[i] * div) % mod;
  }}*/
  /*if (inverse) { xor only
    for (ll i = 0; i < n; i++) {
      a[i] /= n;
  }}*/
}
```

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

```
vector<cplx> mul(vector<cplx>& a, vector<cplx>& b) {
  vector<cplx> c(sz(a)), d(sz(a));
  for (int i = 0; i < sz(b); i++) {
    c[i] = {real(a[i]), real(b[i])};
  }
  c = fft(c);
  for (int i = 0; i < sz(b); i++) {
    int j = (sz(a) - i) % sz(a);
    cplx x = (c[i] + conj(c[j])) / cplx{2, 0}; //fft(a)[i];
    cplx y = (c[i] - conj(c[j])) / cplx{0, 2}; //fft(b)[i];
    d[i] = x * y;
  }
  return fft(d, true);
}
```

4.20 LGS über $\mathbb{R}$

gauss löst LGS $O(n^3)$

```
void normalLine(int line) {
  double factor = mat[line][line];
  for (double& x : mat[line]) x /= factor;
}
void takeAll(int n, int line) {
  for (int i = 0; i < n; i++) {
    if (i == line) continue;
    double diff = mat[i][line];
    for (int j = 0; j <= n; j++) {
      mat[i][j] -= diff * mat[line][j];
}}}
int gauss(int n) {
  vector<bool> done(n, false);
  for (int i = 0; i < n; i++) {
    int swappee = i; // Sucht Pivotzeile für bessere Stabilität.
    for (int j = 0; j < n; j++) {
      if (done[j]) continue;
      if (abs(mat[j][i]) > abs(mat[i][i])) swappee = j;
    }
    swap(mat[i], mat[swappee]);
    if (abs(mat[i][i]) > EPS) {
      normalLine(i);
      takeAll(n, i);
      done[i] = true;
  }}
  // Ab jetzt nur checks bzgl. Eindeutigkeit/Existenz der Lösung.
  for (int i = 0; i < n; i++) {
    bool allZero = true;
    for (int j = i; j < n; j++) allZero &= abs(mat[i][j]) <= EPS;
    if (allZero && abs(mat[i][n]) > EPS) return INCONSISTENT;
    if (allZero && abs(mat[i][n]) <= EPS) return MULTIPLE;
  }
  return UNIQUE;
}
```

4.21 LGS über $\mathbb{F}_p$

gauss löst LGS $O(n^3)$

```
void normalLine(int line, ll p) {
  ll factor = multInv(mat[line][line], p);
  for (ll& x : mat[line]) x = (x * factor) % p;
}
void takeAll(int n, int line, ll p) {
  for (int i = 0; i < n; i++) {
    if (i == line) continue;
    ll diff = mat[i][line];
    for (int j = 0; j <= n; j++) {
      mat[i][j] -= (diff * mat[line][j]) % p;
      mat[i][j] = (mat[i][j] + p) % p;
}}}
void gauss(int n, ll mod) { // Nx(N+1)-Matrix, Körper F_p.
  vector<bool> done(n, false);
  for (int i = 0; i < n; i++) {
    int j = 0;
    while (j < n && (done[j] || mat[j][i] == 0)) j++;
    if (j == n) continue;
    swap(mat[i], mat[j]);
    normalLine(i, mod);
    takeAll(n, i, mod);
    done[i] = true;
}}
// für Eindeutigkeit, Existenz etc. siehe LGS
```