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	$O(\log(n))$

```
struct SegTree {
  using T = ll;
  int n;
  vector<T> tree;
  static constexpr T E = 0; // Neutral element for combine

  SegTree(vector<T>& a) : n(sz(a)), tree(2 * n) {
  //SegTree(int size, T val = E) : n(size), tree(2 * n, val) {
    copy(all(a), tree.begin() + n);
    for (int i = n - 1; i > 0; i--) { // remove for range update
      tree[i] = comb(tree[2 * i], tree[2 * i + 1]);
  }}

  T comb(T a, T b) {return a + b;} // modify this + neutral

  void update(int i, T val) {
    tree[i += n] = val; // apply update code
    while (i /= 2) tree[i] = comb(tree[2 * i], tree[2 * i + 1]);
  }

  T query(int l, int r) {
    T resL = E, resR = E;
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
      if (l&1) resL = comb(resL, tree[l++]);
      if (r&1) resR = comb(tree[--r], resR);
    }
    return comb(resL, resR);
  }

  // OR: range update + point query, needs commutative comb
  void modify(int l, int r, T val) {
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
      if (l&1) tree[l] = comb(tree[l], val), l++;
      if (r&1) --r, tree[r] = comb(tree[r], val);
  }}

  T query(int i) {
    T res = E;
    for (i += n; i > 0; i /= 2) res = comb(res, tree[i]);
    return res;
  }
};
```

1.1.1 Lazy Propagation

Assignment modifications, sum queries

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

```
struct SegTree {
  using T = ll; using U = ll;
  int n;
  static constexpr T E = 0; // Neutral element for combine
  static constexpr U UF = inf; // Unused value by updates
  vector<T> tree;
  int h;
  vector<U> lazy;
  vector<int> k; // size of segments (optional)

  SegTree(const vector<T>& a) : n(sz(a) + 1), tree(2 * n, E),
  //SegTree(int size, T def = E) : n(size + 1), tree(2 * n, def),
      h(__lg(2 * n)), lazy(n, UF), k(2 * n, 1) {
    copy(all(a), tree.begin() + n);
    for (int i = n - 1; i > 0; i--) {
      k[i] = 2 * k[2 * i];
      tree[i] = comb(tree[2 * i], tree[2 * i + 1]);
  }}

  T comb(T a, T b) {return a + b;} // Modify this + E

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

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

  void push(int i) {
    for (int s = h; s > 0; s--) push_down(i >> s);
  }

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

  void update(int l, int r, U val) {
    l += n, r += n;
    int l0 = l, r0 = r;
    push(l0), push(r0 - 1);
    for (; l < r; l /= 2, r /= 2) {
      if (l&1) apply(l++, val);
      if (r&1) apply(--r, val);
    }
    build(l0), build(r0 - 1);
  }

  T query(int l, int r) {
    l += n, r += n;
    push(l), push(r - 1);
    T resL = E, resR = E;
    for (; l < r; l /= 2, r /= 2) {
      if (l&1) resL = comb(resL, tree[l++]);
      if (r&1) resR = comb(tree[--r], resR);
    }
    return comb(resL, resR);
  }

  // Optional:
  int lower_bound(int l, int r, T x) {
    l += n, r += n;
    push(l), push(r - 1);
    int a[64] = {}, lp = 0, rp = 64;
    for (; l < r; l /= 2, r /= 2) {
      if (l&1) a[lp++] = l++;
      if (r&1) a[--rp] = --r;
    }
    for (int i : a) if (i != 0 && tree[i] >= x) { // Modify this
      while (i < n) {
        push_down(i);
        if (tree[2 * i] >= x) i = 2 * i; // And this
        else i = 2 * i + 1;
      }
      return i - n;
    }
    return -1;
  }
};
```

1.2 Wavelet Tree

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

```
struct WaveletTree {
  using it = vector<ll>::iterator;
  WaveletTree *ln = nullptr, *rn = nullptr;
  vector<int> b = {0};
  ll lo, hi;

  WaveletTree(vector<ll> in) : WaveletTree(all(in)) {}

  WaveletTree(it from, it to) : // call above one
    lo(*min_element(from, to)), hi(*max_element(from, to) + 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) return;
    it pivot = stable_partition(from, to, f);
    ln = new WaveletTree(from, pivot);
    rn = new WaveletTree(pivot, to);
  }

  // kth element in sort[l, r) all 0-indexed
  ll kth(int l, int r, int k) {
    if (k < 0 || l + k >= r) 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;}
};
```

1.3 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	$O(\log(n))$

```cpp
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	$O(n \cdot \log(n))$
`prefix_sum`	summe von $[0,i]$	$O(\log(n))$
`update`	addiert ein Delta zu allen Elementen $[l,r)$. $l \leq r!$	$O(\log(n))$

```cpp
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;
}
```

1.4 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++)
```

1.5 (Implicit) Treap (Cartesian Tree)

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

```cpp
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
};
```

1.6 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!

```cpp
struct SparseTable {
  vector<vector<int>> st;
  ll *a;
  int better(int lidx, int ridx) {
    return a[lidx] <= a[ridx] ? lidx : ridx;
  }
  void init(vector<ll>* vec) {
    int n = sz(*vec);
    a = vec->data();
    st.assign(__lg(n) + 1, vector<int>(n));
    iota(all(st[0]), 0);
    for (int j = 0; (2 << j) <= n; j++) {
      for (int i = 0; i + (2 << j) <= n; i++) {
        st[j + 1][i] = better(st[j][i] , st[j][i + (1 << j)]);
  }}}
  int queryIdempotent(int l, int r) {
    if (r <= l) return -1;
    int j = __lg(r - l); //31 - __builtin_clz(r - l);
    return better(st[j][l] , st[j][r - (1 << j)]);
  }
};
```

1.7 STL-Bitset

```cpp
bitset<10> bits(0b000010100);
bits._Find_first(); //2
bits._Find_next(2); //4
bits._Find_next(4); //10 bzw. N
bits[x] = 1;        //not bits.set(x) or bits.reset(x)!
bits[x].flip();     //not bits.flip(x)!
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	$O(\log(n))$

```
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);
  }
};
```

1.9 Lichao

```
vector<ll> xs; // IMPORTANT: Initialize before constructing!
int findX(int i) {return lower_bound(all(xs), i) - begin(xs);}
struct Fun { // Default: Linear function. Change as needed.
  ll m, c;
  ll operator()(int x) {return m*xs[x] + c;}
};
// Default: Computes min. Change lines with comment for max.
struct Lichao {
  static constexpr Fun id = {0, inf}; // {0, -inf}
  int n, cap;
  vector<Fun> seg;
  Lichao() : n(sz(xs)), cap(2<<__lg(n)), seg(2*cap, id) {}
  void _insert(Fun f, int l, int r, int i) {
    while (i < 2*cap){
      int m = (l+r)/2;
      if (m >= n) {r = m; i = 2*i; continue;}
      Fun &g = seg[i];
      if (f(m) < g(m)) swap(f, g); // >
      if (f(l) < g(l)) r = m, i = 2*i; // >
      else l = m, i = 2*i+1;
  }}
  void insert(Fun f) {_insert(f, 0, cap, 1);}
  void _segmentInsert(Fun f, int l, int r, int a, int b, int i) {
    if (l <= a && b <= r) _insert(f, a, b, i);
    else if (a < r && l < b){
      int m = (a+b)/2;
      _segmentInsert(f, l, r, a, m, 2*i);
      _segmentInsert(f, l, r, m, b, 2*i+1);
  }}
  void segmentInsert(Fun f, ll l, ll r) {
    _segmentInsert(f, findX(l), findX(r), 0, cap, 1);
  }
  ll _query(int x) {
    ll ans = inf; // -inf
    for (int i = x + cap; i > 0; i /= 2) {
      ans = min(ans, seg[i](x)); // max
    }
    return ans;
  }
  ll query(ll x) {return _query(findX(x));}
};
```

1.10 Policy Based Data Structures

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

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

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
template<typename T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
                  tree_order_statistics_node_update>;
// T.order_of_key(x): number of elements strictly less than x
// *T.find_by_order(k): k-th element
template<typename T>
struct chash {
  static const
          uint64_t C = ll(2e18 * acos(-1)) | 199; // random odd
  size_t operator()(T o) const {
    return __builtin_bswap64(hash<T>()(o) * C);
}};
template<typename K, typename V>
using hashMap = gp_hash_table<K, V, chash<K>>;
template<typename T>
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.

```
// Lower Envelope mit MONOTONEN Inserts und Queries. Jede neue
// Gerade hat kleinere Steigung als alle vorherigen.
struct Line {
  ll m, b;
  ll operator()(ll x) {return m*x+b;}
};
vector<Line> ls;
int ptr = 0;
bool bad(Line l1, Line l2, Line l3) {
  return (l3.b-l1.b)*(l1.m-l2.m) < (l2.b-l1.b)*(l1.m-l3.m);
}
void add(ll m, ll b) { // Laufzeit O(1) amortisiert
  while (sz(ls) > 1 && bad(ls.end()[-2], ls.end()[-1], {m, b})) {
    ls.pop_back();
  }
  ls.push_back({m, b});
  ptr = min(ptr, sz(ls) - 1);
}
ll query(ll x) { // Laufzeit: O(1) amortisiert
  ptr = min(ptr, sz(ls) - 1);
  while (ptr < sz(ls)-1 && ls[ptr + 1](x) < ls[ptr](x)) ptr++;
  return ls[ptr](x);
}
```

```
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)
  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;
  }
};
```

1.12 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·findSet + n·unionSets`	Folge von Befehlen	$O(n+m\cdot\alpha(n))$

```
// unions[i] >= 0 => unions[i] =  parent
// unions[i]  < 0 => unions[i] = -size
vector<int> unions;
void init(int n) { //Initialisieren
  unions.assign(n, -1);
}
int findSet(int a) { // Pfadkompression
  if (unions[a] < 0) return a;
  return unions[a] = findSet(unions[a]);
}
void linkSets(int a, int b) { // Union by size.
  if (unions[b] > unions[a]) swap(a, b);
  unions[b] += unions[a];
  unions[a] = b;
}
void unionSets(int a, int b) { // Diese Funktion aufrufen.
  if (findSet(a) != findSet(b)) linkSets(findSet(a), findSet(b));
}
int size(int a) {
  return -unions[findSet(a)];
}
```

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	$O(1)$

```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{t+1, T{}}))->second;
  }

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

```cpp
template<typename T>
struct persistentArray {
  int time;
  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 Kruskal

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

```cpp
sort(all(edges));
vector<Edge> mst;
ll 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;
}}
```

2.2 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 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])`.

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

void dfs_sz(int v = 0, int from = -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]); //changes adj!
}}}

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

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

template<typename F>
void for_intervals(int u, int v, F&& f) {
  for (;; v = par[nxt[v]]) {
    if (in[v] < in[u]) swap(u, v);
    f(max(in[u], in[nxt[v]]), in[v] + 1);
    if (in[nxt[v]] <= in[u]) return;
}}

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

2.4 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	$O(1)$

```cpp
struct LCA {
  vector<ll> depth;
  vector<int> visited, first;
  int idx;
  SparseTable st; //sparse table Seite 2

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

  void dfs(vector<vector<int>>& adj, int v, ll d=0) {
    visited[idx] = v, depth[idx] = d;
    first[v] = min(idx, first[v]), idx++;

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

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

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

2.5 Centroids

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

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

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

pair<int, int> find_centroid(int root) {
  s.resize(sz(adj));
  dfs_sz(root);
  return dfs_cent(root, -1, s[root]);
}
```

2.6 Eulertouren

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

```cpp
vector<vector<int>> idx;
vector<int> to, validIdx, cycle;
vector<bool> used;

void addEdge(int u, int v) {
  idx[u].push_back(sz(to));
  to.push_back(v);
  used.push_back(false);
  idx[v].push_back(sz(to)); // für ungerichtet
  to.push_back(u);
  used.push_back(false);
}

void euler(int v) { // init idx und validIdx
  for (;validIdx[v] < sz(idx[v]); validIdx[v]++) {
    if (!used[idx[v][validIdx[v]]]) {
      int u = to[idx[v][validIdx[v]]];
      used[idx[v][validIdx[v]]] = true;
      used[idx[v][validIdx[v]] ^ 1] = true; // für ungerichtet
      euler(u);
    }}
  cycle.push_back(v); // 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>> 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 $O(|V|\cdot\log(|V|))$

```cpp
vector<vector<int>> adj;
map<vector<int>, int> known; // dont reset!
int treeLabel(int v, int from = -1) {
  vector<int> children;
  for (int u : adj[v]) {
    if (u == from) continue;
    children.push_back(treeLabel(u, v));
  }
  sort(all(children));
  if (known.find(children) == known.end()) {
    known[children] = sz(known);
  }
  return known[children];
}
```

2.8 Kürzeste Wege

2.8.1 Bellmann-Ford-Algorithmus

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

```cpp
auto bellmannFord(int n, vector<edge>& edges, int start) {
  vector<ll> dist(n, INF), prev(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;
        prev[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; //return prev?
}
```

2.8.2 Algorithmus von Dijkstra

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

```cpp
using path = pair<ll, int>; //dist, destination
auto dijkstra(const vector<vector<path>>& adj, int start) {
  priority_queue<path, vector<path>, greater<path>> pq;
  vector<ll> dist(sz(adj), INF);
  vector<int> prev(sz(adj), -1);
  dist[start] = 0; pq.emplace(0, start);

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

    for (auto [du, u] : adj[v]) {
      ll newDist = dv + du;
      if (newDist < dist[u]) {
        dist[u] = newDist;
        prev[u] = v;
        pq.emplace(dist[u], u);
  }}}
  return dist; //return prev;
}
```

2.8.3 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 ⇔ `dist[i][i] < 0`.

```cpp
vector<vector<ll>> dist; // Entfernung zwischen je zwei Punkten
vector<vector<int>> next;
void floydWarshall() {
  next.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) {
        next[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++) {
        // only needed if dist can be negative
        if (dist[i][k] == INF || dist[k][j] == INF) continue;
        if (dist[i][j] > dist[i][k] + dist[k][j]) {
          dist[i][j] = dist[i][k] + dist[k][j];
          next[i][j] = next[i][k];
}}}}}

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

2.8.4 Matrix-Algorithmus

Sei d_{ij} die Distanzmatrix von G, dann gibt d^k_{ij} 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^k_{ij} 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 Kannte ein,`id`=delete Zeitpunkt	$O(\log(n))$
`eraseEdge`	entfernt Kante `id`	$O(\log(n))$

```cpp
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 u, int v) {
    return lct.connected(&lct.nodes[u], &lct.nodes[v]);
  }
  void addEdge(int u, int v, int id) {
    lct.nodes[id + n] = LCT::Node(id + n, id + n);
    edges[id] = {u, v};
    if (connected(u, v)) {
      int old = lct.query(&lct.nodes[u], &lct.nodes[v]);
      if (old < id) eraseEdge(old);
    }
    if (!connected(u, v)) {
      lct.link(&lct.nodes[u], &lct.nodes[id + n]);
      lct.link(&lct.nodes[v], &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]);
  }}
};
```

2.10 Erdős-Gallai

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

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

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

2.11 Strongly Connected Components (Tarjan)

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

```
vector<vector<int>> adj, sccs;
int counter;
vector<bool> inStack;
vector<int> low, idx, s; //idx enthält Index der SCC pro Knoten.
void visit(int v) {
  int old = low[v] = counter++;
  s.push_back(v); inStack[v] = true;
  for (auto u : adj[v]) {
    if (low[u] < 0) visit(u);
    if (inStack[u]) low[v] = min(low[v], low[u]);
  }
  if (old == low[v]) {
    sccs.push_back({});
    for (int u = -1; u != v;) {
      u = s.back(); s.pop_back(); inStack[u] = false;
      idx[u] = sz(sccs) - 1;
      sccs.back().push_back(u);
}}}
void scc() {
  inStack.assign(sz(adj), false);
  low.assign(sz(adj), -1);
  idx.assign(sz(adj), -1);
  sccs.clear();
  counter = 0;
  for (int i = 0; i < sz(adj); i++) {
    if (low[i] < 0) visit(i);
}}
```

2.12 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`

2.13 Artikulationspunkte, Brücken und BCC

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

Wichtig: isolierte Knoten und Brücken sind keine BCC.

```
vector<vector<Edge>> adj;
vector<int> num;
int counter, rootCount, root;
vector<bool> isArt;
vector<Edge> bridges, st;
vector<vector<Edge>> bcc;
int dfs(int v, int from = -1) {
  int me = num[v] = ++counter, top = me;
  for (Edge& e : adj[v]) {
    if (e.id == from){}
    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(adj), 0);
  isArt.assign(sz(adj), false);
  bridges.clear();
  st.clear();
  bcc.clear();
  for (int v = 0; v < sz(adj); v++) {
    if (!num[v]) {
      root = v;
      rootCount = 0;
      dfs(v);
      isArt[v] = rootCount > 1;
}}}
```

2.14 2-SAT

```
struct sat2 {
  int n; // + scc variablen
  vector<int> sol;
  sat2(int vars) : n(vars*2), adj(n) {}
  static int var(int i) {return i << 1;} // use this!
  void addImpl(int a, int b) {
    adj[a].push_back(b);
    adj[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 solve() {
    scc(); //scc code von oben
    sol.assign(n, -1);
    for (int i = 0; i < n; i += 2) {
      if (idx[i] == idx[i + 1]) return false;
      sol[i] = idx[i] < idx[i + 1];
      sol[i + 1] = !sol[i];
    }
    return true;
  }
};
```

2.15 Maximal Cliques

`bronKerbosch` berechnet alle maximalen Cliquen $O(3^{\frac{n}{3}})$

`addEdge` fügt **ungerichtete** Kante ein $O(1)$

```
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 = min(P._Find_first(), 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}, {});
}
```

2.16 Cycle Counting

`findBase` berechnet Basis $O(|V| \cdot |E|)$
`count` zählt Zykel $O\left(2^{|base|}\right)$

- jeder Zyklus ist das xor von einträgen in `base`.

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

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

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

  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 v = 0, int from = -1, cycle cur = {}) {
    if (adj.empty()) return;
    if (seen[v]) {
      addBase(cur ^ paths[v]);
    } else {
      seen[v] = true;
      paths[v] = cur;
      for (auto [u, id] : adj[v]) {
        if (u == from) continue;
        cur[id].flip();
        findBase(u, v, cur);
        cur[id].flip();
    }}}

  //cycle must be constrcuted from base
  bool isCycle(cycle cur) {
    if (cur.none()) return false;
    init(sz(adj)); // union find Seite 4
    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;
  }
};
```

2.17 Wert des maximalen Matchings

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

```cpp
constexpr int MOD=1'000'000'007, I=10;
vector<vector<ll>> adj, mat;
int max_matching() {
  int ans = 0;
  mat.assign(sz(adj), {});
  for (int _ = 0; _ < I; _++) {
    for (int v = 0; v < sz(adj); v++) {
      mat[v].assign(sz(adj), 0);
      for (int u : adj[v]) {
        if (u < v) {
          mat[v][u] = rand() % (MOD - 1) + 1;
          mat[u][v] = MOD - mat[v][u];
    }}}
    gauss(sz(adj), MOD); //LGS Seite 17
    int rank = 0;
    for (auto& row : mat) {
      if (*max_element(all(row)) != 0) rank++;
    }
    ans = max(ans, rank / 2);
  }
  return ans;
}
```

2.18 Allgemeines maximales Matching

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

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

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

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

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

  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(adj)) 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 v) {
    label[v] = {sz(adj), -1};
    first[v] = sz(adj);
    head = tail = 0;
    for (que[tail++] = v; head < tail;) {
      int x = que[head++];
      for (int y : adj[x]) {
        if (pairs[y] == sz(adj) && y != v) {
          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 v = 0; v < sz(adj); v++) {
      if (pairs[v] < sz(adj) || !augment(v)) continue;
      matching++;
      for (int i = 0; i < tail; i++)
        label[que[i]] = label[pairs[que[i]]] = {-1, -1};
      label[sz(adj)] = {-1, -1};
    }
    return matching;
  }
};
```

2.19 Rerooting Template

```cpp
// Usual Tree DP can be broken down in 4 steps:
// - Initialize dp[v] = identity
// - Iterate over all children w and take a value for w
//   by looking at dp[w] and possibly the edge label of v -> w
// - combine the values of those children
//   usually this operation should be commutative and associative
// - finalize the dp[v] after iterating over all children
struct Reroot {
  using T = ll;
  // identity element
  T E() {}
  // x: dp value of child
  // e: index of edge going to child
  T takeChild(T x, int e) {}
  T comb(T x, T y) {}
  // called after combining all dp values of children
  T fin(T x, int v) {}
  vector<vector<pair<int, int>>> g;
  vector<int> ord, pae;
  vector<T> dp;
  T dfs(int v) {
    ord.push_back(v);
    for (auto [w, e] : g[v]) {
      g[w].erase(find(all(g[w]), pair(v, e^1)));
      pae[w] = e^1;
      dp[v] = comb(dp[v], takeChild(dfs(w), e));
    }
    return dp[v] = fin(dp[v], v);
  }
  vector<T> solve(int n, vector<pair<int, int>> edges) {
    g.resize(n);
    for (int i = 0; i < n-1; i++) {
      g[edges[i].first].emplace_back(edges[i].second, 2*i);
      g[edges[i].second].emplace_back(edges[i].first, 2*i+1);
    }
    pae.assign(n, -1);
    dp.assign(n, E());
    dfs(0);
    vector<T> updp(n, E()), res(n, E());
    for (int v : ord) {
      vector<T> pref(sz(g[v])+1), suff(sz(g[v])+1);
      if (v != 0) pref[0] = takeChild(updp[v], pae[v]);
      for (int i = 0; i < sz(g[v]); i++){
        auto [u, w] = g[v][i];
        pref[i+1] = suff[i] = takeChild(dp[u], w);
        pref[i+1] = comb(pref[i], pref[i+1]);
      }
      for (int i = sz(g[v])-1; i >= 0; i--) {
        suff[i] = comb(suff[i], suff[i+1]);
      }
      for (int i = 0; i < sz(g[v]); i++) {
        updp[g[v][i].first] = fin(comb(pref[i], suff[i+1]), v);
      }
      res[v] = fin(pref.back(), v);
    }
    return res;
  }
};
```

2.20 Virtual Trees

```cpp
// needs dfs in- and out- time and lca function
vector<int> in, out;
void virtualTree(vector<int> ind) { // indices of used nodes
  sort(all(ind), [&](int x, int y) {return in[x] < in[y];});
  for (int i = 0, n = sz(ind); i < n - 1; i++) {
    ind.push_back(lca(ind[i], ind[i + 1]));
  }
  sort(all(ind), [&](int x, int y) {return in[x] < in[y];});
  ind.erase(unique(all(ind)), ind.end());
  int n = ind.size();
  vector<vector<int>> tree(n);
  vector<int> st = {0};
  for (int i = 1; i < n; i++) {
    while (in[ind[i]] >= out[ind[st.back()]]) st.pop_back();
    tree[st.back()].push_back(i);
    st.push_back(i);
  }
  // virtual directed tree with n nodes, original indices in ind
  // weights can be calculated, e.g. with binary lifting
}
```

2.21 Maximum Cardinatlity 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

```cpp
vector<vector<int>> adj;
vector<int> pairs; // Der gematchte Knoten oder -1.
vector<bool> visited;
bool dfs(int v) {
  if (visited[v]) return false;
  visited[v] = true;
  for (int u : adj[v]) if (pairs[u] < 0 || dfs(pairs[u])) {
    pairs[u] = v; pairs[v] = u; return true;
  }
  return false;
}
int kuhn(int l) { // l = #Knoten links.
  pairs.assign(sz(adj), -1);
  int ans = 0;
  // Greedy Matching. Optionale Beschleunigung.
  for (int v = 0; v < l; v++) for (int u : adj[v])
    if (pairs[u] < 0) {pairs[u] = v; pairs[v] = u; ans++; break;}
  for (int v = 0; v < l; v++) if (pairs[v] < 0) {
    visited.assign(l, false);
    ans += dfs(v);
  }
  return ans; // Größe des Matchings.
}
```

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

```cpp
vector<vector<int>> adj;
// pairs ist der gematchte Knoten oder -1
vector<int> pairs, dist, ptr;
bool bfs(int l) {
  queue<int> q;
  for(int v = 0; v < l; v++) {
    if (pairs[v] < 0) {dist[v] = 0; q.push(v);}
    else dist[v] = -1;
  }
  bool exist = false;
  while(!q.empty()) {
    int v = q.front(); q.pop();
    for (int u : adj[v]) {
      if (pairs[u] < 0) {exist = true; continue;}
      if (dist[pairs[u]] < 0) {
        dist[pairs[u]] = dist[v] + 1;
        q.push(pairs[u]);
  }}}
  return exist;
}
bool dfs(int v) {
  for (; ptr[v] < sz(adj[v]); ptr[v]++) {
    int u = adj[v][ptr[v]];
    if (pairs[u] < 0 ||
       (dist[pairs[u]] > dist[v] && dfs(pairs[u]))) {
      pairs[u] = v; pairs[v] = u;
      return true;
  }}
  return false;
}
int hopcroft_karp(int l) { // l = #Knoten links
  int ans = 0;
  pairs.assign(sz(adj), -1);
  dist.resize(l);
  // Greedy Matching, optionale Beschleunigung.
  for (int v = 0; v < l; v++) for (int u : adj[v])
    if (pairs[u] < 0) {pairs[u] = v; pairs[v] = u; ans++; break;}
  while(bfs(l)) {
    ptr.assign(l, 0);
    for(int v = 0; v < l; v++) {
      if (pairs[v] < 0) ans += dfs(v);
  }}
  return ans;
}
```

2.22 Global Mincut

`stoer_wagner` berechnet globalen Mincut $O\left(|V||E| + |V|^2 \cdot \log(|E|)\right)$

`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.

```cpp
struct Edge {
  int from, to;
  ll cap;
};
```

```
vector<vector<Edge>> adj, tmp;
vector<bool> erased;
void merge(int u, int v) {
  tmp[u].insert(tmp[u].end(), all(tmp[v]));
  tmp[v].clear();
  erased[v] = true;
  for (auto& vec : tmp) {
    for (Edge& e : vec) {
      if (e.from == v) e.from = u;
      if (e.to == v) e.to = u;
}}}
ll stoer_wagner() {
  ll res = INF;
  tmp = adj;
  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;
}
```

2.23 Max-Flow

2.24 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>> adj;
  vector<int> pref, con;
  vector<ll> dist;
  const int s, t;
  ll maxflow, mincost;

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

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

  bool SPFA() {
    pref.assign(sz(adj), -1);
    dist.assign(sz(adj), INF);
    vector<bool> inqueue(sz(adj));
    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 : adj[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(adj), 0);
    maxflow = mincost = 0;
    while (SPFA()) extend();
  }
};
```

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 $O(1)$

```
struct Edge {
  int to, rev;
  ll f, c;
};
vector<vector<Edge>> adj;
int s, t;
vector<int> pt, dist;
void addEdge(int u, int v, ll c) {
  adj[u].push_back({v, (int)sz(adj[v]), 0, c});
  adj[v].push_back({u, (int)sz(adj[u]) - 1, 0, 0});
}
bool bfs(ll lim) {
  dist.assign(sz(adj), -1);
  dist[s] = 0;
  queue<int> q({s});
  while (!q.empty() && dist[t] < 0) {
    int v = q.front(); q.pop();
    for (Edge& e : adj[v]) {
      if (dist[e.to] < 0 && e.c - e.f >= lim) {
        dist[e.to] = dist[v] + 1;
        q.push(e.to);
  }}}
  return dist[t] >= 0;
}
bool dfs(int v, ll flow) {
  if (v == t) return true;
  for (; pt[v] < sz(adj[v]); pt[v]++) {
    Edge& e = adj[v][pt[v]];
    if (dist[e.to] != dist[v] + 1) continue;
    if (e.c - e.f >= flow && dfs(e.to, flow)) {
      e.f += flow;
      adj[e.to][e.rev].f -= flow;
      return true;
  }}
  return false;
}
ll maxFlow(int source, int target) {
  s = source, t = target;
  ll flow = 0;
  for (ll lim = (1LL << 62); lim >= 1; lim /= 2) {
    while (bfs(lim)) {
      pt.assign(sz(adj), 0);
      while (dfs(s, lim)) flow += lim;
  }}
  return flow;
}
```

2.25 Maximum Weight Bipartite Matching

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

```cpp
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);
  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++) {
    vector<int> aug(r, -1);
    vector<bool> s(l);
    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 (aug[yy] < 0 && slack[yy].first < delta) {
          tie(delta, x) = slack[yy];
          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 (aug[y] >= 0) ly[y] += delta;
          else slack[y].first -= delta;
        }}
      aug[y] = x;
      x = yx[y];
      if (x < 0) break;
      s[x] = true;
      for (int y = 0; y < r; y++) {
        if (aug[y] < 0) {
          double alt = lx[x] + ly[y] - costs[x][y];
          if (slack[y].first > alt) {
            slack[y] = {alt, x};
    }}}}
    while (y >= 0) {
      yx[y] = aug[y];
      swap(y, xy[aug[y]]);
  }}
  return accumulate(all(lx), 0.0) +
         accumulate(all(ly), 0.0); // Wert des Matchings
}
```

3 Geometrie

3.1 Closest Pair

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

```cpp
ll rec(vector<pt>::iterator a, int l, int r) {
  if (r - l < 2) return INF;
  int m = (l + r) / 2;
  ll midx = a[m].real();
  ll ans = min(rec(a, l, m), rec(a, m, r));
  inplace_merge(a+l, a+m, a+r, [](const pt& x, const pt& y) {
    return x.imag() < y.imag();
  });
  pt tmp[8];
  fill(all(tmp), a[l]);
  for (int i = l + 1, next = 0; i < r; i++) {
    if (ll x = a[i].real() - midx; x * x < ans) {
      for (pt& p : tmp) ans = min(ans, norm(p - a[i]));
      tmp[next++ & 7] = a[i];
    }
  }
  return ans;
}

ll shortestDist(vector<pt> a) { // sz(pts) > 1
  sort(all(a), [](const pt& x, const pt& y) {
    return x.real() < y.real();
  });
  return rec(a.begin(), 0, sz(a));
}
```

3.2 Konvexehülle

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

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

```cpp
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));
  auto half = [&](auto begin, auto end, int t) {
    for (auto it = begin; it != end; it++) {
      while (k > t && cross(h[k-2], h[k-1], *it) <= 0) k--;
      h[k++] = *it;
  }};
  half(all(pts), 1);// Untere Hülle.
  half(next(pts.rbegin()), pts.rend(), k);// Obere Hülle.
  h.resize(k);
  return h;
}
```

3.3 Rotating calipers

`antipodalPoints` berechnet antipodale Punkte $O(n)$
WICHTIG: Punkte müssen gegen den Uhrzeigersinn sortiert sein und konvexes Polygon bilden!

```cpp
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;
}
```

3.4 Formeln – `std::complex`

```cpp
// 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;
}

// charakterisiert 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;
}
```

```cpp
// 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 || b-a > EPS || c < -EPS || c-a > 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 Geraden durch a-b. 2d und 3d
double distToLine(pt a, pt b, pt p) {
  return abs(cross(p - a, b - a)) / abs(b - a);
}
// Projiziert 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)};
}
```

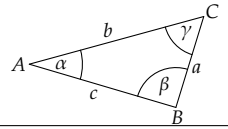
```cpp
// Liegt p auf der Strecke a-b?
bool pointOnLineSegment(pt a, pt b, pt p) {
  if (cross(a, b, p) != 0) return false;
  double dist = norm(a - b);
  return norm(a - p) <= dist && norm(b - p) <= dist;
}
// Entfernung von Punkt p zur Strecke a-b.
double distToSegment(pt a, pt b, pt p) {
  if (a == 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);
  });
}
```

```cpp
bool left(pt p) {return real(p) < 0 ||
                        (real(p) == 0 && imag(p) < 0);}

void sortAround(pt p, vector<pt>& ps) {
  sort(all(ps), [&](const pt& a, const pt& b){
    if (left(a - p) != left(b - p))
      return left(a - p) > left(b - p);
    return cross(p, a, b) > 0;
  });
}
```

Generell:

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



$\beta = 90°$:

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

```cpp
// 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);
}
// Zentrum des Kreises durch alle Eckpunkte
// a, b und c nicht kollinear
pt circumCenter(pt a, pt b, pt c) {
  b -= a, c -= a;
  pt d = b * norm(c) - c * norm(b);
  d = {-d.imag(), d.real()};
  return a + d / cross(b, c) / 2.0;
}
//  1 => p außerhalb Kreis durch a,b,c
//  0 => p auf Kreis durch a,b,c
// -1 => p im Kreis durch a,b,c
int insideOutCenter(pt a, pt b, pt c, pt p) {// braucht lll
  return sgn(imag((c-b)*conj(p-c)*(a-p)*conj(b-a)));
}
// Sind die Dreiecke a1, b1, c1, and a2, b2, c2 ähnlich?
// Erste Zeile testet Ähnlichkeit mit gleicher Orientierung,
// zweite Zeile testet Ähnlichkeit mit verschiedener Orientierung
bool similar(pt a1, pt b1, pt c1, pt a2, pt b2, pt c2) {
  return (b2-a2) * (c1-a1) == (b1-a1) * (c2-a2) ||
         (b2-a2) * conj(c1-a1) == conj(b1-a1) * (c2-a2);
}
```

```cpp
// Flächeninhalt eines Polygons (nicht selbstschneidend).
// Punkte gegen den Uhrzeigersinn: positiv, sonst negativ.
double area(const vector<pt>& poly) { //poly[0] == poly.back()
  double res = 0;
  for (int i = 0; i + 1 < sz(poly); i++)
    res += cross(poly[i], poly[i + 1]);
  return 0.5 * res;
}
// Anzahl drehungen einer Polyline um einen Punkt
// p nicht auf rand und poly[0] == poly.back()
// res != 0 or (res & 1) != 0 um inside zu prüfen bei
// selbstschneidenden Polygonen (definitions Sache)
ll windingNumber(pt p, const vector<pt>& poly) {
  ll res = 0;
  for (int i = 0; i + 1 < sz(poly); i++) {
    pt a = poly[i], b = poly[i + 1];
    if (real(a) > real(b)) swap(a, b);
    if (real(a) <= real(p) && real(p) < real(b) &&
        cross(p, a, b) < 0) {
      res += orientation(p, poly[i], poly[i + 1]);
  }}
  return res;
}
// Testet, ob ein Punkt im Polygon liegt (beliebige Polygone).
// Ändere Zeile 32 falls rand zählt, poly[0] == poly.back()
bool inside(pt p, const vector<pt>& poly) {
  bool in = false;
  for (int i = 0; i + 1 < sz(poly); i++) {
    pt a = poly[i], b = poly[i + 1];
    if (pointOnLineSegment(a, b, p)) return false;
```

```
    if (real(a) > real(b)) swap(a,b);
    if (real(a) <= real(p) && real(p) < real(b) &&
        cross(p, a, b) < 0) {
      in ^= 1;
  }}
  return in;
}
// convex hull without duplicates, h[0] != h.back()
// apply comments if border counts as inside
bool inside(pt p, const vector<pt>& hull) {
  int l = 0, r = sz(hull) - 1;
  if (cross(hull[0], hull[r], p) >= 0) return false; // > 0
  while (l + 1 < r) {
    int m = (l + r) / 2;
    if (cross(hull[0], hull[m], p) > 0) l = m; // >= 0
    else r = m;
  }
  return cross(hull[l], hull[r], p) > 0; // >= 0
}
void rotateMin(vector<pt>& hull) {
  auto mi = min_element(all(hull), [](const pt& a, const pt& b){
    return real(a) == real(b) ? imag(a) < imag(b)
                              : real(a) < real(b);
  });
  rotate(hull.begin(), mi, hull.end());
}
// convex hulls without duplicates, h[0] != h.back()
vector<pt> minkowski(vector<pt> ps, vector<pt> qs) {
  rotateMin(ps);
  rotateMin(qs);
  ps.push_back(ps[0]);
  qs.push_back(qs[0]);
  ps.push_back(ps[1]);
  qs.push_back(qs[1]);
  vector<pt> res;
  for (ll i = 0, j = 0; i + 2 < sz(ps) || j + 2 < sz(qs);) {
    res.push_back(ps[i] + qs[j]);
    auto c = cross(ps[i + 1] - ps[i], qs[j + 1] - qs[j]);
    if(c >= 0) i++;
    if(c <= 0) j++;
  }
  return res;
}
// convex hulls without duplicates, h[0] != h.back()
double dist(const vector<pt>& ps, const vector<pt>& qs) {
  for (pt& q : qs) q *= -1;
  auto p = minkowski(ps, qs);
  p.push_back(p[0]);
  double res = 0.0;
  //bool intersect = true;
  for (ll i = 0; i + 1 < sz(p); i++) {
    //intersect &= cross(p[i], p[i+1] - p[i]) <= 0;
    res = max(res, cross(p[i], p[i+1]-p[i]) / abs(p[i+1]-p[i]));
  }
  return res;
}
```

```
bool left(pt of, pt p) {return cross(p, of) < 0 ||
                               (cross(p, of) == 0 && dot(p, of) > 0);}
// convex hulls without duplicates, hull[0] == hull.back() and
// hull[0] must be a convex point (with angle < pi)
// returns index of corner where dot(dir, corner) is maximized
int extremal(const vector<pt>& hull, pt dir) {
  dir *= pt(0, 1);
  int l = 0, r = sz(hull) - 1;
  while (l + 1 < r) {
    int m = (l + r) / 2;
    pt dm = hull[m+1]-hull[m];
    pt dl = hull[l+1]-hull[l];
    if (left(dl, dir) != left(dl, dm)) {
      if (left(dl, dm)) l = m;
      else r = m;
    } else {
      if (cross(dir, dm) < 0) l = m;
      else r = m;
  }}
  return r;
}
// convex hulls without duplicates, hull[0] == hull.back() and
// hull[0] must be a convex point (with angle < pi)
// {} if no intersection
// {x} if corner is only intersection
// {a, b} segments (a,a+1) and (b,b+1) intersected (if only the
// border is intersected corners a and b are the start and end)
vector<int> intersect(const vector<pt>& hull, pt a, pt b) {
  int endA = extremal(hull, (a-b) * pt(0, 1));
  int endB = extremal(hull, (b-a) * pt(0, 1));
  // cross == 0 => line only intersects border
  if (cross(hull[endA], a, b) > 0 ||
      cross(hull[endB], a, b) < 0) return {};
  int n = sz(hull) - 1;
  vector<int> res;
  for (auto _ : {0, 1}) {
    int l = endA, r = endB;
    if (r < l) r += n;
    while (l + 1 < r) {
      int m = (l + r) / 2;
      if (cross(hull[m % n], a, b) <= 0 &&
          cross(hull[m % n], a, b) != hull(poly[endB], a, b))
        l = m;
      else r = m;
    }
    if (cross(hull[r % n], a, b) == 0) l++;
    res.push_back(l % n);
    swap(endA, endB);
    swap(a, b);
  }
  if (res[0] == res[1]) res.pop_back();
  return res;
}
```

```
// 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

```
// 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);
}
```

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

3.6 Half-plane intersection

```cpp
constexpr ll inf = 0x1FFF'FFFF'FFFF'FFFF;//THIS CODE IS WIP
bool left(pt p) {return real(p) < 0 ||
            (real(p) == 0 && imag(p) < 0);}
struct hp {
  pt from, to;
  hp(pt a, pt b) : from(a), to(b) {}
  hp(pt dummy) : hp(dummy, dummy) {}
  bool dummy() const {return from == to;}
  pt dir() const {return dummy() ? to : to - from;}
  bool operator<(const hp& o) const {
    if (left(dir()) != left(o.dir()))
      return left(dir()) > left(o.dir());
    return cross(dir(), o.dir()) > 0;
  }
  using lll = __int128;
  using ptl = complex<lll>;
  ptl mul(lll m, ptl p) const {return m*p;}//ensure 128bit
  bool check(const hp& a, const hp& b) const {
    if (dummy() || b.dummy()) return false;
    if (a.dummy()) {
      ll ort = sgn(cross(b.dir(), dir()));
      if (ort == 0) return cross(from, to, a.from) < 0;
      return cross(b.dir(), a.dir()) * ort > 0;
    }
    ll y = cross(a.dir(), b.dir());
    ll z = cross(b.from - a.from, b.dir());
    ptl i = mul(y, a.from) + mul(z, a.dir()); //intersect a and b
    // check if i is outside/right of x
    return imag(conj(mul(sgn(y),dir()))*(i-mul(y,from))) < 0;
  }
};
constexpr ll lim = 2e9+7;
deque<hp> intersect(vector<hp> hps) {
  hps.push_back(hp(pt{lim+1,-1}));
  hps.push_back(hp(pt{lim+1,1}));
  sort(all(hps));
  deque<hp> dq = {hp(pt{-lim, 1})};
  for (auto x : hps) {
    while (sz(dq) > 1 && x.check(dq.end()[-1], dq.end()[-2]))
      dq.pop_back();
    while (sz(dq) > 1 && x.check(dq[0], dq[1]))
      dq.pop_front();
    if (cross(x.dir(), dq.back().dir()) == 0) {
      if (dot(x.dir(), dq.back().dir()) < 0) return {};
      if (cross(x.from, x.to, dq.back().from) < 0)
        dq.pop_back();
      else continue;
    }
    dq.push_back(x);
  }
  while (sz(dq) > 2 && dq[0].check(dq.end()[-1], dq.end()[-2]))
    dq.pop_back();
  while (sz(dq) > 2 && dq.end()[-1].check(dq[0], dq[1]))
    dq.pop_front();
  if (sz(dq) < 3) return {};
  return dq;
}
```

4 Mathe

4.1 Longest Increasing Subsequence

- lower_bound ⇒ streng monoton
- upper_bound ⇒ monoton

```cpp
vector<int> lis(vector<ll>& a) {
  int n = sz(a), len = 0;
  vector<ll> dp(n, INF), dp_id(n), prev(n);
  for (int i = 0; i < n; i++) {
    int pos = lower_bound(all(dp), a[i]) - dp.begin();
    dp[pos] = a[i];
    dp_id[pos] = i;
    prev[i] = pos ? dp_id[pos - 1] : -1;
    len = max(len, pos + 1);
  }
  // reconstruction
  vector<int> res(len);
  for (int x = dp_id[len-1]; len--; x = prev[x]) {
    res[len] = x;
  }
  return res; // indices of one LIS
}
```

4.2 Zykel Erkennung

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

```cpp
pair<ll, ll> 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++;
  }
  return {start, length};
}
```

4.3 Permutationen

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

```cpp
vector<ll> kthperm(ll n, ll k) {
  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 $(res \in [0,n!))$ $O(n \cdot \log(n))$

```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 (int i = sz(v); i > 0; i--) {
    res = res * i + v[i - 1];
  }
  return res;
}
```

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

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

```cpp
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 $a^b \bmod n$ $O(\log(b))$

```cpp
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

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

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

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 α und β mit $\alpha x+\beta m=1$.
- Nach Kongruenz gilt $\alpha x+\beta m \equiv \alpha x \equiv 1 \bmod m$.
- $x^{-1} := \alpha \bmod m$

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

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

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 $(\overline{x},\overline{y})$ 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.

```cpp
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 $x \equiv a \bmod n$, $x \equiv b \bmod m$:

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

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.

```cpp
struct CRT {
  using lll = __int128;
  lll M = 1, sol = 0; // Solution unique modulo M
  bool hasSol = true;

  // Adds congruence x = a (mod m)
  void add(ll a, ll m) {
    auto [d, s, t] = extendedEuclid(M, m);
    if((a - sol) % d != 0) hasSol = false;
    lll z = M/d * s;
    M *= m/d;
    sol = (z % M * (a-sol) % M + sol + M) % M;
  }
};
```

4.9 Primzahltest & Faktorisierung

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

```cpp
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 = ((lll)v * v) % n;
      if (v == n - 1 || v <= 1) break;
    }
    if (v != n - 1) return false;
  }
  return true;
}
```

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

```cpp
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, i = n/2 + 7;
  auto f = [&](lll c){return (c * c + i) % n;};
  for (ll t = 30; 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 $O(\sqrt[3]{n})$

```cpp
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 Matrix-Exponent

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

`calc` berechnet $m^b_{y,x}$ $O(\log(b) \cdot n^2)$

```cpp
vector<mat> pows;
void precalc(mat m) {
  pows = {mat(1), m};
  for (int i = 1; i < 60; i++) pows.push_back(pows[i] * pows[i]);
}
ll calc(int x, int y, ll b) {
  vector<ll> v(pows[0].m.size());
  v[x] = 1;
  for (ll i = 1; b > 0; i++) {
    if (b & 1) v = pows[i] * v;
    b /= 2;
  }
  return v[y];
}
```

4.12 Lineare Rekurrenz

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

```cpp
constexpr ll mod = 1'000'000'007;
vector<ll> BerlekampMassey(const vector<ll>& s) {
  int n = sz(s), L = 0, m = 0;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;

  ll b = 1;
  for (int i = 0; i < n; i++) {
    m++;
    ll d = s[i] % mod;
    for (int j = 1; j <= L; j++) {
      d = (d + C[j] * s[i - j]) % mod;
    }
    if (!d) continue;
    T = C;
    ll coef = d * powMod(b, mod-2, mod) % mod;
    for (int j = m; j < n; j++) {
      C[j] = (C[j] - coef * B[j - m]) % mod;
    }
    if (2 * L > i) continue;
    L = i + 1 - L;
    swap(B, T);
    b = d;
    m = 0;
  }

  C.resize(L + 1);
  C.erase(C.begin());
  for (auto& x : C) x = (mod - x) % mod;
  return C;
}
```

Sei $f(n) = c_{n-1}f(n-1) + c_{n-2}f(n-2) + \cdots + c_0 f(0)$ eine lineare Rekurrenz.

`kthTerm` Berechnet k-ten Term einer Rekurrenz n-ter Ordnung $O(\log(k) \cdot n^2)$

```cpp
constexpr ll mod = 1'000'000'007;
vector<ll> modMul(const vector<ll>& a, const vector<ll>& b,
                  const vector<ll>& c) {
  ll n = sz(c);
  vector<ll> res(n * 2 + 1);
  for (int i = 0; i <= n; i++) { //a*b
    for (int j = 0; j <= n; j++) {
      res[i + j] += a[i] * b[j];
      res[i + j] %= mod;
  }}
  for (int i = 2 * n; i > n; i--) { //res%c
    for (int j = 0; j < n; j++) {
      res[i - 1 - j] += res[i] * c[j];
      res[i - 1 - j] %= mod;
  }}
  res.resize(n + 1);
  return res;
}
ll kthTerm(const vector<ll>& f, const vector<ll>& c, ll k) {
  assert(sz(f) == sz(c));
  vector<ll> tmp(sz(c) + 1), a(sz(c) + 1);
  tmp[0] = a[1] = 1; //tmp = (x^k) % c
  for (k++; k > 0; k /= 2) {
    if (k & 1) tmp = modMul(tmp, a, c);
    a = modMul(a, a, c);
  }
  ll res = 0;
  for (int i = 0; i < sz(c); i++) res += (tmp[i+1] * f[i]) % mod;
  return res % mod;
}
```

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

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

4.13 Diskreter Logarithmus

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

```cpp
ll dlog(ll a, ll b, ll m) { //a > 0!
  ll bound = sqrtl(m) + 1; //memory usage bound < p
  vector<pair<ll, ll>> vals(bound);
  for (ll i = 0, e = 1; i < bound; i++, e = (e * a) % m) {
    vals[i] = {e, i};
  }
  vals.emplace_back(m, 0);
  sort(all(vals));
  ll fact = powMod(a, m - bound - 1, m);

  for (ll i = 0; i < m; i += bound, b = (b * fact) % m) {
    auto it = lower_bound(all(vals), pair<ll, ll>{b, 0});
    if (it->first == b) {
      return (i + it->second) % m;
  }}
  return -1;
}
```

4.14 Diskrete Quadratwurzel

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

Wichtig: p muss prim sein!

```cpp
ll sqrtMod(ll a, ll p) {// teste mit legendre ob lösung existiert
  if (a < 2) return a;
  ll t = 0;
  while (legendre((t*t-4*a) % p, p) >= 0) t = rng() % p;
  ll b = -t, c = -t, d = 1, m = p;
  for (m++; m /= 2; b = (a+a-b*b) % p, a = (a*a) % p) {
    if (m % 2) {
      d = (c-d*b) % p;
      c = (c*a) % p;
    } else {
      c = (d*a - c*b) % p;
  }}
  return (d + p) % p;
}
```

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) $O(|ans| \cdot \log(\varphi(n)) \cdot \log(n))$

```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);
  for (ll res = 1; res < n; res++) // oder zufällige Reihenfolge
    if (isPrimitive(res, n, phin, phiFacs)) return res;
  return -1;
}
```

4.16 Diskrete n-te Wurzel

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

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

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

4.17 Legendre-Symbol

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

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

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

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

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

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

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 $\text{ggT}(a,b) = 1$.
$\Rightarrow$ 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`!

```cpp
constexpr ll N = 10'000'000;
ll small[N], power[N], sieved[N];
vector<ll> primes;

//wird aufgerufen mit (p^k, p, k) für prime p und k > 0
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 p;}

void sieve() { // O(N)
  small[1] = power[1] = sieved[1] = 1;
  for (ll i = 2; i < N; i++) {
    if (small[i] == 0) {
      primes.push_back(i);
      for (ll pk = i, k = 1; pk < N; pk *= i, k++) {
        small[pk] = i;
        power[pk] = pk;
        sieved[pk] = mu(pk, i, k); // Aufruf ändern!
    }}
    for (ll j=0; i*primes[j] < N && primes[j] < small[i]; j++) {
      ll k = i * primes[j];
      small[k] = power[k] = primes[j];
      sieved[k] = sieved[i] * sieved[primes[j]];
    }
    if (i * small[i] < N && power[i] != i) {
      ll k = i * small[i];
      small[k] = small[i];
      power[k] = power[i] * small[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!
  }}
  if (n > 1) res *= mu(n, n, 1);
  return res;
}
```

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 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**: $a^m \equiv a \pmod{m}$

4.19 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 $O(1)$

```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;
}}}}
```

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

4.21 LGS über $\mathbb{F}_p$

`gauss` löst LGS $O(n^3)$

```cpp
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 < sz(mat[i]); j++) {
      mat[i][j] -= (diff * mat[line][j]) % p;
      mat[i][j] = (mat[i][j] + p) % p;
}}}

void gauss(int n, ll mod) {
  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 über R Seite 17
```

4.22 LGS über $\mathbb{R}$

`gauss` löst LGS $O(n^3)$

```cpp
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 < sz(mat[i]); 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;
}
```

4.23 Numerisch Extremstelle bestimmen

```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;
}
```

4.24 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);
}
```

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

```using cplx = complex<double>;
void fft(vector<cplx>& a, bool inv = false) {
  int n = sz(a);
  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]);
  }
  static vector<cplx> ws(2, 1);
  for (static int k = 2; k < n; k *= 2) {
    ws.resize(n);
    cplx w = polar(1.0, acos(-1.0) / k);
    for (int i=k; i<2*k; i++) ws[i] = ws[i/2] * (i % 2 ? w : 1);
  }
  for (int s = 1; s < n; s *= 2) {
    for (int j = 0; j < n; j += 2 * s) {
      for (int k = 0; k < s; k++) {
        cplx u = a[j + k], t = a[j + s + k];
        t *= (inv ? conj(ws[s + k]) : ws[s + k]);
        a[j + k] = u + t;
        a[j + s + k] = u - t;
        if (inv) a[j + k] /= 2, a[j + s + k] /= 2;
}}}}
```

```constexpr ll mod = 998244353, root = 3;
void ntt(vector<ll>& a, bool inv = false) {
  int n = sz(a);
  auto b = a;
  ll r = inv ? powMod(root, mod - 2, mod) : root;
  for (int s = n / 2; s > 0; s /= 2) {
    ll ws = powMod(r, (mod - 1) / (n / s), mod), w = 1;
    for (int j = 0; j < n / 2; j += s) {
      for (int k = j; k < j + s; k++) {
        ll u = a[j + k], t = a[j + s + k] * w % mod;
        b[k] = (u + t) % mod;
        b[n/2 + k] = (u - t + mod) % mod;
      }
      w = w * ws % mod;
    }
    swap(a, b);
  }
  if (inv) {
    ll div = powMod(n, mod - 2, mod);
    for (auto& x : a) x = x * div % mod;
}}
```

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

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

```vector<cplx> mul(vector<ll>& a, vector<ll>& b) {
  int n = 1 << (__lg(sz(a) + sz(b) - 1) + 1);
  vector<cplx> c(all(a)), d(n);
  c.resize(n);
  for (int i = 0; i < sz(b); i++) c[i] = {real(c[i]), b[i]};
  fft(c);
  for (int i = 0; i < n; i++) {
    int j = (n - i) & (n - 1);
    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;
  }
  fft(d, true);
  return d;
}
```

4.26 Operations on Formal Power Series

```vector<ll> poly_inv(const vector<ll>& a, int n) {
  vector<ll> q = {powMod(a[0], mod-2, mod)};
  for (int len = 1; len < n; len *= 2){
    vector<ll> a2 = a, q2 = q;
    a2.resize(2*len), q2.resize(2*len);
    ntt(q2);
    for (int j : {0, 1}) {
      ntt(a2);
      for (int i = 0; i < 2*len; i++) a2[i] = a2[i]*q2[i] % mod;
      ntt(a2, true);
      for (int i = 0; i < len; i++) a2[i] = 0;
    }
    for (int i = len; i < min(n, 2*len); i++) {
      q.push_back((mod - a2[i]) % mod);
  }}
  return q;
}
vector<ll> poly_deriv(vector<ll> a) {
  for (int i = 1; i < sz(a); i++)
    a[i-1] = a[i] * i % mod;
  a.pop_back();
  return a;
}
vector<ll> poly_integr(vector<ll> a) {
  if (a.empty()) return {0};
  a.push_back(a.back() * powMod(sz(a), mod-2, mod) % mod);
  for (int i = sz(a)-2; i > 0; i--)
    a[i] = a[i-1] * powMod(i, mod-2, mod) % mod;
  a[0] = 0;
  return a;
}
vector<ll> poly_log(vector<ll> a, int n) {
  a = mul(poly_deriv(a), poly_inv(a, n));
  a.resize(n-1);
  a = poly_integr(a);
  return a;
}
vector<ll> poly_exp(vector<ll> a, int n) {
  vector<ll> q = {1};
  for (int len = 1; len < n; len *= 2) {
    vector<ll> p = poly_log(q, 2*len);
    for (int i = 0; i < 2*len; i++)
      p[i] = (mod - p[i] + (i < sz(a) ? a[i] : 0)) % mod;
    vector<ll> q2 = q;
    q2.resize(2*len);
    ntt(p), ntt(q2);
    for (int i = 0; i < 2*len; i++) p[i] = p[i] * q2[i] % mod;
    ntt(p, true);
    for (int i = len; i < min(n, 2*len); i++) q.push_back(p[i]);
  }
  return q;
}
```

4.27 Inversionszahl

```cpp
ll inversions(const vector<ll>& v) {
  Tree<pair<ll, ll>> t; //ordered statistics tree Seite 4
  ll res = 0;
  for (ll i = 0; i < sz(v); i++) {
    res += i - t.order_of_key({v[i], i});
    t.insert({v[i], i});
  }
  return res;
}
```

4.28 Satz von Sprague-Grundy

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

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

X ist genau dann gewonnen, wenn $g(X) > 0$ ist.
Wenn man k Spiele in den Zuständen $X_1,\ldots,X_k$ hat, dann ist die Grundy-Zahl des Gesamtzustandes $g(X_1) \oplus \ldots \oplus g(X_k)$.

4.29 Kombinatorik

Wilsons Theorem A number n is prime if and only if $(n-1)! \equiv -1 \bmod n$.
(n is prime if and only if $(m-1)! \cdot (n-m)! \equiv (-1)^m \bmod n$ for all m in $\{1,\ldots,n\}$)

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

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

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

Binomialkoeffizienten Die Anzahl der k-elementigen Teilmengen einer n-elementigen Menge.
`precalc` berechnet $n!$ und $n!^{-1}$ vor $O(lim)$
`calc_binom` berechnet Binomialkoeffizient $O(1)$

```cpp
constexpr ll lim = 10'000'000;
ll fac[lim], inv[lim];

void precalc() {
  fac[0] = inv[0] = 1;
  for (int i = 1; i < lim; i++) fac[i] = fac[i-1] * i % mod;
  inv[lim - 1] = multInv(fac[lim - 1], mod);
  for (int i = lim - 1; i > 0; i--) inv[i-1] = inv[i] * i % mod;
}
ll calc_binom(ll n, ll k) {
  if (n < 0 || n < k || k < 0) return 0;
  return (inv[k] * inv[n-k] % mod) * fac[n] % mod;
}
```

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

`calc_binom` berechnet Binomialkoeffizient ($n \le 61$) $O(k)$

```cpp
ll calc_binom(ll n, ll k) {
  if (k > n) return 0;
  ll r = 1;
  for (ll d = 1; d <= k; d++) {// Reihenfolge => Teilbarkeit
    r *= n--, r /= d;
  }
  return r;
}
```

`calc_binom` berechnet Binomialkoeffizient modulo Primzahl p $O(p-n)$

```cpp
ll calc_binom(ll n, ll k, ll p) {
  assert(n < p); //wichtig: sonst falsch!
  if (k > n) return 0;
  ll x = k % 2 != 0 ? p-1 : 1;
  for (ll c = p-1; c > n; c--) {
    x *= c - k; x %= p;
    x *= multInv(c, p); x %= p;
  }
  return x;
}
```

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.

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

- 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 (^{k} beginnen.

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

Euler-Zahlen 1. Ordnung Die Anzahl der Permutationen von $\{1,\ldots,n\}$ mit genau k Anstiegen. Für die n-te Zahl gibt es n mögliche Positionen zum Einfügen. Dabei wird entweder ein Anstieg in zwei gesplitted oder ein Anstieg um n ergänzt.

$$\left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1 \qquad \left\langle {n \atop k} \right\rangle = (k+1)\left\langle {n-1 \atop k} \right\rangle + (n-k)\left\langle {n-1 \atop k-1} \right\rangle = \sum_{i=0}^{k} (-1)^i \binom{n+1}{i} (k+1-i)^n$$

- 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 $\{1,1,\ldots,n,n\}$ mit genau k Anstiegen.

$$\left\langle\left\langle {n \atop 0} \right\rangle\right\rangle = 1 \qquad \left\langle\left\langle {n \atop n} \right\rangle\right\rangle = 0 \qquad \left\langle\left\langle {n \atop k} \right\rangle\right\rangle = (k+1)\left\langle\left\langle {n-1 \atop k} \right\rangle\right\rangle + (2n-k-1)\left\langle\left\langle {n-1 \atop k-1} \right\rangle\right\rangle$$

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

Stirling-Zahlen 1. Ordnung Die Anzahl der Permutationen von $\{1,\ldots,n\}$ mit genau k Zyklen. Es gibt zwei Möglichkeiten für die n-te Zahl. Entweder sie bildet einen eigene Zyklus, oder sie kann an jeder Position in jedem Zyklus einsortiert werden.

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

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

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

- Berechne Polynom mit FFT und benutzte 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 nichtleere Teilmengen zu zerlegen. Es gibt k Möglichkeiten die n in eine $n-1$-Partition einzuordnen. Dazu kommt der Fall, dass die n in ihrer eigenen Teilmenge (alleine) steht.

$$\left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1 \qquad \left\{ {n \atop k} \right\} = k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\} = \frac{1}{k!}\sum_{i=0}^{k} (-1)^{k-i}\binom{k}{i} i^n$$

- 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 $\{1,\ldots,n\}$. Wie Stirling-Zahlen 2. Ordnung ohne Limit durch k.

$$B_1 = 1 \qquad B_n = \sum_{k=0}^{n-1} B_k \binom{n-1}{k} = \sum_{k=0}^{n} \left\{ {n \atop k} \right\} \qquad B_{p^m+n} \equiv m \cdot B_n + B_{n+1} \bmod p$$

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

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

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

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

- 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 Twelvefold Way (verteile n Bälle auf k Boxen)

Bälle / Boxen	identisch / identisch	verschieden / identisch	identisch / verschieden	verschieden / verschieden
–	$p_k(n+k)$	$\sum_{i=0}^{k} \left\{ {n \atop i} \right\}$	$\binom{n+k-1}{k-1}$	k^n
Bälle pro Box ≥ 1	$p_k(n)$	$\left\{ {n \atop k} \right\}$	$\binom{n-1}{k-1}$	$k!\left\{ {n \atop k} \right\}$
Bälle pro Box ≤ 1	$[n \le k]$	$[n \le k]$	$\binom{k}{n}$	$n!\binom{k}{n}$

[Bedingung]: `return Bedingung ? 1 : 0;`

Binomialkoeffizienten

$$\frac{n!}{k!(n-k)!} = \binom{n}{k} = \binom{n}{n-k} = \frac{n}{k}\binom{n-1}{k-1} = \frac{n-k+1}{k}\binom{n}{k-1} = \binom{n-1}{k}+\binom{n-1}{k-1} = (-1)^k\binom{k-n-1}{k} \approx 2^n\cdot\frac{2}{\sqrt{2\pi n}}\cdot\exp\left(-\frac{2(x-\frac{n}{2})^2}{n}\right)$$

$$\sum_{k=0}^{n}\binom{n}{k}=2^n \qquad \sum_{k=0}^{n}\binom{k}{m}=\binom{n+1}{m+1} \qquad \sum_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n} \qquad \sum_{k=0}^{n}\binom{r+k}{k}=\binom{r+n+1}{n}$$

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

Important Numbers

10^x	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	7 560	64	−27	+7	1 229	5	
5	83 160	128	−9	+3	9 592	6	
6	720 720	240	−17	+3	78 498	7	
7	8 648 640	448	−9	+19	664 579	8	
8	73 513 440	768	−11	+7	5 761 455	8	
9	735 134 400	1 344	−63	+7	50 847 534	9	
10	6 983 776 800	2 304	−33	+19	455 052 511	10	
11	97 772 875 200	4 032	−23	+3	4 118 054 813	10	
12	963 761 198 400	6 720	−11	+39	37 607 912 018	11	
13	9 316 358 251 200	10 752	−29	+37	346 065 536 839	12	
14	97 821 761 637 600	17 280	−27	+31	3 204 941 750 802	12	
15	866 421 317 361 600	26 880	−11	+37	29 844 570 422 669	13	
16	8 086 598 962 041 600	41 472	−63	+61	279 238 341 033 925	13	
17	74 801 040 398 884 800	64 512	−3	+3	2 623 557 157 654 233	14	
18	897 612 484 786 617 600	103 680	−11	+3	24 739 954 287 740 860	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

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

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, $k\in\mathbb{N}$)

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

Verschiedenes

Türme von Hanoi, minimale Schirttzahl:	$T_n=2^n-1$
#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	$!n=(n-1)(!(n-1)+!(n-2))=\lfloor\frac{n!}{e}+\frac{1}{2}\rfloor$
	$\lim_{n\to\infty}\frac{!n}{n!}=\frac{1}{e}$

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

Beschreibung	Strategie
$M=[pile_i]$ $[x]:=\{1,\ldots,x\}$	$SG=\oplus_{i=1}^{n} pile_i$ 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 $n\equiv a \bmod (a+1)$ $SG_n=n\%(a+1)\%2$, sonst.
$M_①=\left\lceil\frac{pile_i}{2}\right\rceil$ $M_②=\left\{\left\lceil\frac{pile_i}{2}\right\rceil, pile_i\right\}$	① $SG_{2n}=n$, $SG_{2n+1}=SG_n$ ② $SG_0=0$, $SG_n=\lceil\log_2 n\rceil+1$
$M_①$ = Teiler von $pile_i$ $M_②$ = echte Teiler von $pile_i$	① $SG_0=0$, $SG_n=SG_{②,n}+1$ ② $ST_1=0$, SG_n = #Nullen am Ende von n_{bin}
$M_①=[k]$ $M_②=S$, (S endlich) $M_③=S\cup\{pile_i\}$	$SG_{①,n}=n \bmod (k+1)$ ❶ Niederlage bei $SG=0$ ❷ Niederlage bei $SG=1$ $SG_{③,n}=SG_{②,n}+1$
Für jedes endliche M ist SG eines Stapels irgendwann periodisch.	
Moore's Nim: Beliebige Zahl von maximal k Stapeln.	❶ Schreibe $pile_i$ binär. Addiere ohne Übertrag zur Basis $k+1$. Niederlage, falls Ergebnis gleich 0. ❷ Wenn alle Stapel 1 sind: Niederlage, wenn $n\equiv 1 \bmod (k+1)$. 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 $n\equiv 1,2 \bmod 4$ $SG_n=n+1$, falls $n\equiv 3 \bmod 4$ $SG_n=n-1$, falls $n\equiv 0 \bmod 4$
Kayles' Nim: Zwei mögliche Züge: 1) Nehme beliebige Zahl. 2) Teile Stapel in zwei Stapel (mit Entnahme).	Berechne SG_n für kleine n rekursiv. $n\in[72,83]$: 4,1,2,8,1,4,7,2,1,8,2,7 Periode ab $n=72$ der Länge 12.

5 Strings

5.1 Knuth-Morris-Pratt-Algorithmus

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

```
vector<int> kmpPreprocessing(const string& sub) {
  vector<int> b(sz(sub) + 1);
  b[0] = -1;
  for (int i = 0, j = -1; i < sz(sub);) {
    while (j >= 0 && sub[i] != sub[j]) j = b[j];
    b[++i] = ++j;
  }
  return b;
}
vector<int> kmpSearch(const string& s, const string& sub) {
  vector<int> result, pre = kmpPreprocessing(sub);
  for (int i = 0, j = 0; i < sz(s);) {
    while (j >= 0 && s[i] != sub[j]) j = pre[j];
    i++; j++;
    if (j == sz(sub)) {
      result.push_back(i - j);
      j = pre[j];
  }}
  return result;
}
```

5.2 Z-Algorithmus

z_i := Längstes gemeinsames Präfix von $s_0 \cdots s_{n-1}$ und $s_i \cdots s_{n-1}$ $O(n)$
Suchen: Z-Algorithmus auf `P$S` ausführen, Positionen mit $z_i = |P|$ zurückgeben

```
vector<int> Z(const string& s) {
  int n = sz(s);
  vector<int> z(n);
  for (int i = 1, x = 0; i < n; i++) {
    z[i] = max(0, min(z[i - x], x + z[x] - i));
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      x = i, z[i]++;
  }}
  return z;
}
```

5.3 Rolling Hash

```
// M = 1.7e9 + 9, 1e18L + 9, 2.2e18L + 7
struct Hash {
  static constexpr ll M = 3e18L + 37;
  static constexpr ll Q = 318LL << 53; // Random in [SIGMA+1, M)
  vector<ll> pref = {0}, power = {1};

  Hash(const string& s) {
    for (auto c : s) { // c > 0
      pref.push_back((mul(pref.back(), Q) + c + M) % M);
      power.push_back(mul(power.back(), Q));
  }}

  ll operator()(int l, int r) {
    return (pref[r] - mul(power[r-l], pref[l]) + M) % M;
  }

  static ll mul(__int128 a, ll b) {return a * b % M;}
};
```

5.4 Pattern Matching mit Wildcards

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

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

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

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 $O(n)$

```
vector<int> manacher(const string& t) {
  //transforms "aa" to ".a.a." to find even length palindromes
  string s(sz(t) * 2 + 1, '.');
  for (int i = 0; i < sz(t); i++) s[2 * i + 1] = t[i];

  int mid = 0, r = 0, n = sz(s);
  vector<int> pal(n);
  for (int i = 1; i < n - 1; i++) {
    if (r > i) pal[i] = min(r - i, pal[2 * mid - i]);
    while (pal[i] < min(i, n - i - 1) &&
           s[i + pal[i] + 1] == s[i - pal[i] - 1]) {
      pal[i]++;
    }
    if (i + pal[i] > r) mid = i, r = i + pal[i];
  }

  //convert lengths to constructed string s (optional)
  //for (int i = 0; i < n; i++) pal[i] = 2 * pal[i] + 1;
  return pal;
}
```

5.6 Longest Common Subsequence

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

```
string lcss(const string& a, const string& b) {
  vector<vector<int>> m(sz(a) + 1, vector<int>(sz(b) + 1));
  for (int i = sz(a) - 1; i >= 0; i--) {
    for (int j = sz(b) - 1; j >= 0; j--) {
      if (a[i] == b[j]) m[i][j] = 1 + m[i+1][j+1];
      else m[i][j] = max(m[i+1][j], m[i][j+1]);
  }} // Für die Länge: return m[0][0];
  string res;
  for (int j = 0, i = 0; j < sz(b) && i < sz(a);) {
    if (a[i] == b[j]) res += a[i++], j++;
    else if (m[i][j+1] > m[i+1][j]) j++;
    else i++;
  }
  return res;
}
```

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

```
constexpr ll ALPHABET_SIZE = 26, OFFSET = 'a';
struct AhoCorasick {
  struct vert {
    int suffix = 0, ch, cnt = 0;
    array<int, ALPHABET_SIZE> nxt = {};

    vert(int p, int c) : suffix(-p), ch(c) {}
  };
  vector<vert> aho = {{0, -1}};

  int addString(string &s) {
    int v = 0;
    for (auto c : s) {
      int idx = c - OFFSET;
      if (!aho[v].nxt[idx]) {
        aho[v].nxt[idx] = sz(aho);
        aho.emplace_back(v, idx);
      }
      v = aho[v].nxt[idx];
    }
    aho[v].cnt++;
    return v; // trie node index of pattern (pattern state)
  }

  int getSuffix(int v) {
    if (aho[v].suffix < 0) {
      aho[v].suffix = go(getSuffix(-aho[v].suffix), aho[v].ch);
    }
    return aho[v].suffix;
  }

  int go(int v, int idx) { // Root is v=0, idx is char - OFFSET
    if (aho[v].nxt[idx]) return aho[v].nxt[idx];
    else return v == 0 ? 0 : go(getSuffix(v), idx);
  }

  vector<vector<int>> adj;
  vector<ll> dp;
  void buildGraph() {
    adj.resize(sz(aho));
    dp.assign(sz(aho), 0);
    for (int i = 1; i < sz(aho); i++) {
      adj[getSuffix(i)].push_back(i);
  }}

  void dfs(int v = 0) { // dp on tree
    for (int u : adj[v]) {
      //dp[u] = dp[v] + aho[u].cnt; // pattern count
      dfs(u);
      dp[v] += dp[u]; // no of matches
  }}
};
```

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	$O(n)$

```
bool next(string& s, int maxLen, char mi = '0', char ma = '1') {
  for (int i = sz(s), j = sz(s); i < maxLen; i++)
    s.push_back(s[i % j]);
  while(!s.empty() && s.back() == ma) s.pop_back();
  if (s.empty()) {
    s = mi;
    return false;
  } else {
    s.back()++;
    return true;
}}
```

```
vector<pair<int, int>> duval(const string& s) {
  vector<pair<int, int>> res;
  for (int i = 0; i < sz(s);) {
    int j = i + 1, k = i;
    for (; j < sz(s) && s[k] <= s[j]; j++) {
      if (s[k] < s[j]) k = i;
      else k++;
    }
    while (i <= k) {
      res.push_back({i, i + j - k});
      i += j - k;
  }}
  return res;
}
int minrotation(const string& s) {
  auto parts = duval(s+s);
  for (auto [l, r] : parts) {
    if (l < sz(s) && r >= sz(s)) {
      return l;
}}}
```

- **De-Bruijn-Sequenze $B(\Sigma,n)$:** 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 $\frac{(k!)^{k^{n-1}}}{k^n}$ verschiedene $B(\Sigma,n)$
- $B(\Sigma,n)$ hat Länge $|\Sigma|^n$

`deBruijn`	berechnet ein festes $B(\Sigma,n)$	$O(\lvert\Sigma\rvert^n)$

```
string deBruijn(int n, char mi = '0', char ma = '1') {
  string res, c(1, mi);
  do {
    if (n % sz(c) == 0) res += c;
  } while(next(c, n, mi, ma));
  return res;
}
```

5.9 Suffix-Array

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

```
constexpr int MAX_CHAR = 256;
struct SuffixArray {
  int n;
  vector<int> SA, LCP;
  vector<vector<int>> P;

  SuffixArray(const string& s) : n(sz(s)), SA(n), LCP(n),
    P(__lg(2 * n - 1) + 1, vector<int>(n)) {
    P[0].assign(all(s));
    iota(all(SA), 0);
    sort(all(SA), [&](int a, int b) {return s[a] < s[b];});
    vector<int> x(n);
    for (int k = 1, c = 1; c < n; k++, c *= 2) {
      iota(all(x), n - c);
      for (int ptr = c; int i : SA) if (i >= c) x[ptr++] = i - c;

      vector<int> cnt(k == 1 ? MAX_CHAR : n);
      for (int i : P[k-1]) cnt[i]++;
      partial_sum(all(cnt), begin(cnt));
      for (int i : x | views::reverse) SA[--cnt[P[k-1][i]]] = i;

      auto p = [&](int i) {return i < n ? P[k-1][i] : -1;};
      for (int i = 1; i < n; i++) {
        int a = SA[i-1], b = SA[i];
        P[k][b] = P[k][a] + (p(a) != p(b) || p(a+c) != p(b+c));
    }}
    for (int i = 1; i < n; i++) LCP[i] = lcp(SA[i-1], SA[i]);
  }
  int lcp(int x, int y) {//x & y are text-indices, not SA-indices
    if (x == y) return n - x;
    int res = 0;
    for (int i = sz(P) - 1; i >= 0 && max(x, y) + res < n; i--) {
      if (P[i][x + res] == P[i][y + res]) res |= 1 << i;
    }
    return res;
  }
};
```

5.10 Suffix-Baum

`SuffixTree`	berechnet einen Suffixbaum	$O(\lvert s\rvert)$
`extend`	fügt den nächsten Buchstaben aus s ein	$O(1)$

```
struct SuffixTree {
  struct Vert {
    int start, end, suf; //s[start...end) along parent edge
    map<char, int> nxt;
  };
  string s;
  int needsSuffix, pos, remainder, curVert, curEdge, curLen;
  // Each Vertex gives its children range as [start, end)
  vector<Vert> tree = {Vert{-1, -1, 0, {}}};

  SuffixTree(const string& s_) : s(s_) {
    needsSuffix = remainder = curVert = curEdge = curLen = 0;
    pos = -1;
    for (int i = 0; i < sz(s); i++) extend();
  }
  int newVert(int start, int end) {
    tree.push_back({start, end, 0, {}});
    return sz(tree) - 1;
  }
  void addSuffixLink(int vert) {
    if (needsSuffix) tree[needsSuffix].suf = vert;
    needsSuffix = vert;
  }
  bool fullImplicitEdge(int vert) {
    int len = min(tree[vert].end, pos + 1) - tree[vert].start;
    if (curLen >= len) {
      curEdge += len;
      curLen -= len;
      curVert = vert;
      return true;
    } else {
      return false;
  }}
  void extend() {
    pos++;
    needsSuffix = 0;
    remainder++;
    while (remainder) {
      if (curLen == 0) curEdge = pos;
      if (!tree[curVert].nxt.count(s[curEdge])) {
        int leaf = newVert(pos, sz(s));
        tree[curVert].nxt[s[curEdge]] = leaf;
        addSuffixLink(curVert);
      } else {
        int nxt = tree[curVert].nxt[s[curEdge]];
        if (fullImplicitEdge(nxt)) continue;
        if (s[tree[nxt].start + curLen] == s[pos]) {
          curLen++;
          addSuffixLink(curVert);
          break;
        }
        int split = newVert(tree[nxt].start,
                            tree[nxt].start + curLen);
        tree[curVert].nxt[s[curEdge]] = split;
        int leaf = newVert(pos, sz(s));
        tree[split].nxt[s[pos]] = leaf;
        tree[nxt].start += curLen;
        tree[split].nxt[s[tree[nxt].start]] = nxt;
        addSuffixLink(split);
      }
      remainder--;
      if (curVert == 0 && curLen) {
        curLen--;
        curEdge = pos - remainder + 1;
      } else {
        curVert = tree[curVert].suf ? tree[curVert].suf : 0;
  }}}
};
```

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.

```cpp
constexpr int ALPHABET_SIZE = 26;
constexpr char OFFSET = 'a';
struct SuffixAutomaton {
  struct State {
    int len, link = -1;
    array<int, ALPHABET_SIZE> nxt; // map if large Alphabet
    State(int l) : len(l) {fill(all(nxt), -1);}
  };
  vector<State> st = {State(0)};
  int cur = 0;
  SuffixAutomaton(const string& s) {
    st.reserve(2 * sz(s));
    for (auto c : s) extend(c - OFFSET);
  }
  void extend(int c) {
    int p = cur;
    cur = sz(st);
    st.emplace_back(st[p].len + 1);
    for (; p != -1 && st[p].nxt[c] < 0; p = st[p].link) {
      st[p].nxt[c] = cur;
    }
    if (p == -1) {
      st[cur].link = 0;
    } else {
      int q = st[p].nxt[c];
      if (st[p].len + 1 == st[q].len) {
        st[cur].link = q;
      } else {
        st.emplace_back(st[p].len + 1);
        st.back().link = st[q].link;
        st.back().nxt = st[q].nxt;
        for (; p != -1 && st[p].nxt[c] == q; p = st[p].link) {
          st[p].nxt[c] = sz(st) - 1;
        }
        st[q].link = st[cur].link = sz(st) - 1;
  }}}
  vector<int> calculateTerminals() {
    vector<int> terminals;
    for (int p = cur; p != -1; p = st[p].link) {
      terminals.push_back(p);
    }
    return terminals;
  }
```

```cpp
  // Pair with start index (in t) and length of LCS.
  pair<int, int> longestCommonSubstring(const string& t) {
    int v = 0, l = 0, best = 0, bestp = -1;
    for (int i = 0; i < sz(t); i++) {
      int c = t[i] - OFFSET;
      while (v > 0 && st[v].nxt[c] < 0) {
        v = st[v].link;
        l = st[v].len;
      }
      if (st[v].nxt[c] >= 0) v = st[v].nxt[c], l++;
      if (l > best) best = l, bestp = i;
    }
    return {bestp - best + 1, best};
  }
};
```

5.12 Trie

```cpp
// Zahlenwerte müssen bei 0 beginnen und zusammenhängend sein.
constexpr int ALPHABET_SIZE = 2;
struct node {
  int words, ends;
  array<int, ALPHABET_SIZE> nxt;
  node() : words(0), ends(0) {fill(all(nxt), -1);}
};
vector<node> trie = {node()};
int traverse(const vector<int>& word, int x) {
  int id = 0;
  for (int c : word) {
    if (id < 0 || (trie[id].words == 0 && x <= 0)) return -1;
    trie[id].words += x;
    if (trie[id].nxt[c] < 0 && x > 0) {
      trie[id].nxt[c] = sz(trie);
      trie.emplace_back();
    }
    id = trie[id].nxt[c];
  }
  trie[id].words += x;
  trie[id].ends += x;
  return id;
}
int insert(const vector<int>& word) {
  return traverse(word, 1);
}
bool erase(const vector<int>& word) {
  int id = traverse(word, 0);
  if (id < 0 || trie[id].ends <= 0) return false;
  traverse(word, -1);
  return true;
}
```

6 Python

6.1 Recursion

```python
import sys
sys.setrecursionlimit(1000_007)
```

6.2 IO

```python
n, m = map(int, input().split())
A = list(map(int, input().split()))
print(n, m, *A)
```

7 Sonstiges

7.1 Compiletime

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

```cpp
template<int N>
struct Table {
  int data[N];
  constexpr Table() : data {} {
    for (int i = 0; i < N; i++) data[i] = i;
}};
constexpr Table<100'000> precalculated;
```

7.2 Timed

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

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

7.3 Bit Operations

Bit an Position j lesen	`(x & (1 << j))!= 0`
Bit an Position j setzten	`x |= (1 << j)`
Bit an Position j löschen	`x &= ~(1 << j)`
Bit an Position j flippen	`x ^= (1 << j)`
Anzahl an führenden nullen ($x \neq 0$)	`__builtin_clzll(x)`
Anzahl an schließenden nullen ($x \neq 0$)	`__builtin_ctzll(x)`
Anzahl an 1 bits	`__builtin_popcountll(x)`
i-te Zahl eines Graycodes	`i ^ (i >> 1)`

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

7.4 Overflow-sichere arithmetische Operationen

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

Addition	`__builtin_saddll_overflow(a, b, &c)`
Subtraktion	`__builtin_ssubll_overflow(a, b, &c)`
Multiplikation	`__builtin_smulll_overflow(a, b, &c)`

7.5 Pragmas

```
#pragma GCC optimize("Ofast")
#pragma GCC optimize ("unroll-loops")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,"
                   "popcnt,abm,mmx,avx,tune=native")
#pragma GCC target("fpmath=sse,sse2") // no excess precision
#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: $A[i-1][j] \le A[i][j] \le A[i][j+1]$
`calc` berechnet das DP $O(n^2)$

```
ll calc(int n, int m, const vector<vector<ll>>& C) {
  vector<vector<ll>> dp(m, vector<ll>(n, inf));
  vector<vector<int>> opt(m, vector<int>(n + 1, n - 1));

  for (int i = 0; i < n; i++) dp[0][i] = C[0][i];
  for (int i = 1; i < m; i++) {
    for (int j = n - 1; j >= 0; --j) {
      opt[i][j] = i == 1 ? 0 : opt[i - 1][j];
      for (int k = opt[i][j]; k <= min(opt[i][j+1], j-1); k++) {
        if (dp[i][j] <= dp[i - 1][k] + C[k + 1][j]) continue;
        dp[i][j] = dp[i - 1][k] + C[k + 1][j];
        opt[i][j] = k;
  }}}
  return dp[m - 1][n - 1];
}
```

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

```
vector<vector<ll>> dp;
vector<vector<ll>> C;

void rec(int i, int j0, int j1, int m0, int m1) {
  if (j1 < j0) return;
  int jmid = (j0 + j1) / 2;

  dp[i][jmid] = inf;
  int bestk = m0;
  for (int k = m0; k < min(jmid, m1 + 1); ++k) {
    if (dp[i - 1][k] + C[k + 1][jmid] < dp[i][jmid]) {
      dp[i][jmid] = dp[i - 1][k] + C[k + 1][jmid];
      bestk = k;
  }}

  rec(i, j0, jmid - 1, m0, bestk);
  rec(i, jmid + 1, j1, bestk, m1);
}
ll calc(int n, int m) {
  dp = vector<vector<ll>>(m, vector<ll>(n, inf));
  for (int i = 0; i < n; i++) dp[0][i] = C[0][i];
  for (int i = 1; i < m; i++) {
    rec(i, 0, n - 1, 0, n - 1);
  }
  return dp[m - 1][n - 1];
}
```

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

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

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

7.7 Parallel Binary Search

```
// Q = # of queries, bucket sort is sometimes faster
vector<int> low(Q, 0), high(Q, MAX_OPERATIONS);
while (true) {
  vector<pair<int, int>> focus;
  for (int i = 0; i < Q; i++) if (low[i] < high[i]) {
    focus.emplace_back((low[i] + high[i]) / 2, i);
  }
  if (focus.empty()) break;
  sort(all(focus));

  // reset simulation
  for (int step = 0; auto [mid, i] : focus) {
    while (step <= mid) {
      // simulation step
      step++;
    }
    if (/* requirement already fulfilled */) high[i] = mid;
    else low[i] = mid + 1;
}} // 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)

```
int rotateLeft(int n) { // Der letzte Überlebende, 1-basiert.
  for (int i = 31; i >= 0; i--) {
    if (n & (1 << i)) {
      n &= ~(1 << i);
      break;
  }}
  n <<= 1; n++; return n;
}
```

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

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

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

7.9 Sonstiges

```
// Alles-Header.
#include <bits/stdc++.h>

// Setzt deutsche Tastaturlayout / toggle mit alt + space
setxkbmap de
setxkbmap de,us -option grp:alt_space_toggle

// Schnelle Ein-/Ausgabe mit cin/cout.
cin.tie(nullptr)->ios::sync_with_stdio(false);

// Set mit eigener Sortierfunktion.
set<point2, decltype(comp)> set1(comp);

// STL-Debugging, Compiler flags.
-D_GLIBCXX_DEBUG
#define _GLIBCXX_DEBUG

// 128-Bit Integer/Float. Muss zum Einlesen/Ausgeben
// in einen int oder long long gecastet werden.
__int128, __float128

// float mit Decimaldarstellung
#include <decimal/decimal>
std::decimal::decimal128

// 1e18 < INF < Max_Value / 2
constexpr ll INF = 0x3FFF'FFFF'FFFF'FFFFll;
// 1e9 < INF < Max_Value / 2
constexpr int INF = 0x3FFF'FFFF;
```

7.10 Stress Test

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

7.11 Gemischtes

- **(Minimum) Flow mit Demand d:** Erstelle neue Quelle s' und Senke t' und setzte die folgenden Kapazitäten:
$$c'(s',v)=\sum_{u\in V} d(u,v) \qquad c'(v,t')=\sum_{u\in V} d(v,u)$$
$$c'(u,v)=c(u,v)-d(u,v) \qquad c'(t,s)=x$$
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, ∞ sonst).
- **Johnsons Reweighting Algorithmus:** Initialisiere alle Entfernungen mit `d[i] = 0`. Berechne mit Bellmann-Ford kürzeste Entfernungen. Falls es einen negativen Zyklus gibt abrechen. 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 $a-b\le c$. 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` $\to$ `A` mit Gewicht `w(A)` und Kanten `B` $\to$ `t` mit Gewicht `w(B)` hinzu. Füge Kanten mit Kapazität ∞ von `A` nach `B` hinzu, wo im originalen Graphen Kanten waren. Max-Flow ist die Lösung.
Im Residualgraphen:
  - Das Vertex-Cover sind die Knoten inzident zu den Brücken. *oder*
  - Die Knoten in `A`, die *nicht* von `s` 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 π eine Permutation der Menge X. Die Elemente von X können mit einer von m Farben gefärbt werden. Die Anzahl der Färbungen, die Fixpunkte von π sind, ist $m^{\#(\pi)}$, wobei $\#(\pi)$ die Anzahl der Zyklen von π ist. Die Anzahl der Färbungen von Objekten einer Menge X mit m Farben unter einer Symmetriegruppe G is gegeben durch:
$$[X/G]=\frac{1}{|G|}\sum_{g\in G} m^{\#(g)}$$
- **Verteilung von Primzahlen:** Für alle $n\in\mathbb{N}$ gilt: Ex existiert eine Primzahl p mit $n\le p\le 2n$.
- **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.
- **Dilworths-Theorem:** Sei S eine Menge und $\le$ eine partielle Ordnung (S ist ein Poset). Eine *Kette* ist eine Teilmenge $\{x_1,\dots,x_n\}$ mit $x_1\le\dots\le x_n$. 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 $x\le y$, füge Kante $u_x\to v_y$ hinzu. Wenn Matching zu langsam ist, versuche Struktur des Posets auszunutzen und evtl. anders eine maximale Anitkette zu finden.
- **Turan's-Theorem:** Die Anzahl an Kanten in einem Graphen mit n Knoten der keine clique der größe $x+1$ enthält ist:
$$ext(n,K_{x+1})=\binom{n}{2}-\left[(x-(n \bmod x))\cdot\binom{\lfloor\frac{n}{x}\rfloor}{2}+(n \bmod x)\cdot\binom{\lceil\frac{n}{x}\rceil}{2}\right]$$
- **Euler's-Polyedersatz:** In planaren Graphen gilt $n-m+f-c=1$.
- **Pythagoreische Tripel:** Sei $m>n>0$, $k>0$ und $m\not\equiv n \bmod 2$ dann beschreibt diese Formel alle Pythagoreischen Tripel eindeutig:
$$k\cdot\left(a=m^2-n^2,\quad b=2mn,\quad c=m^2+n^2\right)$$
- **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: $O\left(|V|\cdot\log(|V|)\right)$.
- **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 $O\left(n^2\right)$ zu $O\left(n\log n\right)$ zu verbessern.
- **Mo's Algorithm:** SQRT-Decomposition auf n Intervall Queries $[l,r]$. Gruppiere Queries in $\sqrt{n}$ Blöcke nach linker Grenze l. Sortiere nach Block und bei gleichem Block nach rechter Grenze r. Beantworte Queries offline durch schrittweise Vergrößern/Verkleinern des aktuellen Intervalls. Laufzeit: $O\left(n\cdot\sqrt{n}\right)$. (Anzahl der Blöcke als Konstante in Code schreiben.)
- **SQRT Techniques:**
  - Aufteilen in *leichte* (wert $\le\sqrt{x}$) 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 $\min(w,h)\le\sqrt{X}$.
- **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(1.00000000000001)`?
  - 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++

```cpp
#include <bits/stdc++.h>
using namespace std;

#define tsolve int t; cin >> t; while(t--) solve
#define all(x) ::begin(x), ::end(x)
#define sz(x) (ll)::size(x)

using ll = long long;
using ld = long double;

void solve() {}

int main() {
  cin.tie(0)->sync_with_stdio(false);
  cout << setprecision(16);
  solve();
}
```

8.2 Console

```
alias comp=
      "g++ -std=gnu++17 -O2 -Wall -Wextra -Wconversion -Wshadow"
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:

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

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

```
"\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)
- ~$8e^{-17}$ with excess precision (real value)

```
#include <cfloat>
int main() {
  cout << "Mode: " << FLT_EVAL_METHOD << endl;
  double a = atof("1.2345678");
  double b = a*a;
  cout << b - 1.52415765279683990130 << '\n';
}
```