uni name

team name

Miles Conway, Alex Wang, Yiheng You

2025-09-03

Contest (1)

template.cpp 21 lines

```cpp
#include <bits/stdc++.h>

using namespace std;
using ll = long long;

#define FOR(i, a, b) for (int i = a; i < (b); i++)
#define F0R(i, b) FOR(i, 0, b)
#define all(x) begin(x), end(x)
#define vt vector
#define size(x) ((int) (x).size())
#define ROF(i, a, b) for (int i = (b) - 1; i >= (a); i--)
#define pb push_back
#define f first
#define s second
using vi = vt<int>;

int main() {
    cin.tie(0)->sync_with_stdio(0);
    cin.exceptions(cin.failbit);

}
```

hash.sh 3 lines

```sh
# Hashes a file, ignoring all whitespace and comments. Use for
# verifying that code was correctly typed.
cpp -dD -P -fpreprocessed | tr -d '[:space:]'| md5sum |cut -c-6
```

troubleshoot.txt 52 lines

```
Pre-submit:
Write a few simple test cases if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all data structures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a teammate.
Ask the teammate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a teammate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your teammates think about your algorithm?

Memory limit exceeded:
What is the max amount of memory your algorithm should need?
Are you clearing all data structures between test cases?
```

Mathematics (2)

2.1 Equations

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \Rightarrow \begin{aligned} x &= \frac{ed - bf}{ad - bc} \\ y &= \frac{af - ec}{ad - bc} \end{aligned}$$

2.2 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k - c_1 x^{k-1} - \cdots - c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

2.3 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1$$

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(2n+1)(n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

Data structures (3)

OrderStatisticTree.h

Description: A set (not multiset!) with support for finding the n'th element, and finding the index of an element. To get a map, change `null_type`.
Time: $\mathcal{O}(\log N)$

782797, 16 lines

```cpp
#include <bits/extc++.h>
using namespace __gnu_pbds;

template<class T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
  tree_order_statistics_node_update>;

void example() {
  Tree<int> t, t2; t.insert(8);
  auto it = t.insert(10).first;
  assert(it == t.lower_bound(9));
  assert(t.order_of_key(10) == 1);
  assert(t.order_of_key(11) == 2);
  assert(*t.find_by_order(0) == 8);
  t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

HashMap.h

Description: Hash map with mostly the same API as unordered_map, but ~3x faster. Uses 1.5x memory. Initial capacity must be a power of 2 (if provided).

d77092, 7 lines

```cpp
#include <bits/extc++.h>
// To use most bits rather than just the lowest ones:
struct chash { // large odd number for C
  const uint64_t C = ll(4e18 * acos(0)) | 71;
  ll operator()(ll x) const { return __builtin_bswap64(x*C); }
};
__gnu_pbds::gp_hash_table<ll, int, chash> h({}, {}, {}, {}, {1 << 16})
    ;
```

LazySegtree.h

Description: Generic lazy segment tree
Usage: Implement +, upd and id for Node object; += and id for Lazy object.
Time: $\mathcal{O}(\log N)$.

683133, 49 lines

```cpp
struct LazySeg {
  int n;
  vt<Node> seg;
  vt<Lazy> lazy;
  void init(int _n) {
```

```
    for (n = 1; n < _n; n *= 2);
    seg.resize(2 * n, nid);
    lazy.resize(2 * n, lid);
  }
  void pull(int i) {
    seg[i] = seg[2 * i] + seg[2 * i + 1];
  }
  void push(int i, int l, int r) {
    seg[i].upd(lazy[i], l, r);
    if (r - l > 1) F0R (j, 2) lazy[2 * i + j] += lazy[i];
    lazy[i] = lid;
  }
  void build() {
    for (int i = n - 1; i > 0; i--) pull(i);
  }
  void upd(int lo, int hi, Lazy val) { upd(lo, hi, val, 1, 0, n); }
  void upd(int lo, int hi, Lazy val, int i, int l, int r) {
    if (r == -1) r = n;
    push(i, l, r);
    if (r <= lo || l >= hi) return;
    if (lo <= l && r <= hi) {
      lazy[i] += val;
      push(i, l, r);
      return;
    }
    int m = (l + r) / 2;
    upd(lo, hi, val, 2 * i, l, m);
    upd(lo, hi, val, 2 * i + 1, m, r);
    pull(i);
  }
  Node query() { return query(0, n, 1, 0, n); }
  Node query(int lo, int hi) { return query(lo, hi, 1, 0, n); }
  Node query(int lo, int hi, int i, int l, int r) {
    push(i, l, r);
    if (r <= lo || l >= hi) return nid;
    if (lo <= l && r <= hi) return seg[i];
    int m = (l + r) / 2;
    return query(lo, hi, 2 * i, l, m)
      + query(lo, hi, 2 * i + 1, m, r);
  }
  Node& operator[](int i) {
    return seg[i + n];
  }
};
```

SparseSegtree.h

Description: Generic-ish sparse segment tree (point update, range query).
Usage: Choose appropriate identity element and merge function.
Time: $\mathcal{O}(\log N)$.

d51c9b, 28 lines

```
using ptr = struct Node*;
const int sz = 1 << 30;
struct Node {
  #define func(a, b) min(a, b)
  #define ID INF
  ll val;
  ptr lc, rc;

  ptr get(ptr& p) { return p ? p : p = new Node {ID}; }

  ll query(int lo, int hi, int l = 0, int r = sz) {
    if (lo >= r || hi <= l) return ID;
    if (lo <= l && r <= hi) return val;
    int m = (l + r) / 2;
    return func(get(lc)->query(lo, hi, l, m),
      get(rc)->query(lo, hi, m, r));
  }

  ll upd(int i, ll nval, int l = 0, int r = sz) {
    if (r - l == 1) return val = nval;
    int m = (l + r) / 2;
    if (i < m) get(lc)->upd(i, nval, l, m);
    else get(rc)->upd(i, nval, m, r);
    return val = func(get(lc)->val, get(rc)->val);
  }
  #undef ID
  #undef func
};
```

PersistentSegtree.h

Description: Generic-ish persistent segment tree (point update, range query).
Usage: Choose appropriate identity element and merge function.
Time: $\mathcal{O}(\log N)$.

f155de, 30 lines

```
using ptr = struct Node*;
const int sz = 1 << 18;

struct Node {
  #define func(a, b) min(a, b)
  #define ID inf
  int v;
  ptr lc, rc;

  ptr pull(ptr lc, ptr rc) {
    return new Node {func(lc->v, rc->v), lc, rc};
  }

  ptr upd(int i, int nv, int l = 0, int r = sz) {
    if (r - l == 1) return new Node {nv};
    int m = (l + r) / 2;
    if (i < m) return pull(lc->upd(i, nv, l, m), rc);
    else return pull(lc, rc->upd(i, nv, m, r));
  }

  int query(int lo, int hi, int l = 0, int r = sz) {
    if (lo >= r || hi <= l) return ID;
    if (lo <= l && r <= hi) return v;
    int m = (l + r) / 2;
    return func(lc->query(lo, hi, l, m),
      rc->query(lo, hi, m, r));
  }
  #undef id
  #undef func
};
```

LiChaoTree.h

Description: LiChao tree
Usage: self explanatory i think
Time: $\mathcal{O}(\log N)$.

00d540, 36 lines

```
struct Line {
  ll m, c;
  ll operator()(ll x) {
    return m * x + c;
  }
};

const ll sz = 1ll << 30;

using ptr = struct Node*;
struct Node {
  ptr lc, rc;
  Line line;

  Node(Line _line) {
    line = _line;
    lc = rc = 0;
  }
};

// min tree (flip signs for max)
void add(ptr& n, Line loser, ll l = 0, ll r = sz) {
  if (n ? 0 : n = new Node(loser)) return;
  ll m = (l + r) / 2;
  if (loser(m) < n->line(m)) swap(loser, n->line);
  if (r - l == 1) return;
  if (loser(l) < n->line(l)) add(n->lc, loser, l, m);
  else add(n->rc, loser, m, r);
}

ll query(ptr n, ll x, ll l = 0, ll r = sz) {
  if (!n) return sz;
  ll m = (l + r) / 2;
  if (x < m) return min(n->line(x), query(n->lc, x, l, m));
  else return min(n->line(x), query(n->rc, x, m, r));
}
```

SparseTable.h

Description: Generic sparse table for idempotent operations.
Usage: Define the desired operation
Time: $\mathcal{O}(N \log N)$ build, $\mathcal{O}(1)$ query.

3e1539, 18 lines

```
template<class T> struct RMQ {
  #define func min
  vt<vt<T>> dp;
  void init(const vt<T>& v) {
    dp.resize(32 - __builtin_clz(size(v)), vt<T>(size(v)));
    copy(all(v), begin(dp[0]));
    for (int j = 1; 1 << j <= size(v); ++j) {
      for (int i = 0; i < size(v) - (1 << j) + 1; i++)
        dp[j][i] = func(dp[j - 1][i],
          dp[j - 1][i + (1 << (j - 1))]);
    }
  }
  T query(int l, int r) {
    int d = 31 - __builtin_clz(r - l);
    return func(dp[d][l], dp[d][r - (1 << d)]);
  }
  #undef func
};
```

StaticRangeQuery.h

Description: Generic static range query for associative operations.
Usage: Define the desired operation
Time: $\mathcal{O}(N \log N)$ build, $\mathcal{O}(1)$ query.

82b9ca, 34 lines

```
template<class T> struct RangeQuery {
  #define comb(a, b) (a) + (b)
  #define id 0
  int lg, n;
  vt<vt<T>> stor;
  vt<T> a;
  void fill(int l, int r, int ind) {
    if (ind < 0) return;
    int m = (l + r) / 2;
    T prod = id;
    FOR (i, m, r) stor[i][ind] = prod = comb(prod, a[i]);
    prod = id;
    ROF (i, l, m) stor[i][ind] = prod = comb(a[i], prod);
    fill(l, m, ind - 1);
    fill(m, r, ind - 1);
  }
  template <typename It>
  void build(It l, It r) {
    lg = 1;
    while ((1 << lg) < r - l) lg++;
```

```
    n = 1 << lg;
    a.resize(n, id);
    for (It i = l; i != r; i++) a[i - l] = *i;
    stor.resize(n, vt<T>(32 - __builtin_clz(n)));
    fill(0, n, lg - 1);
  }
  T query(int l, int r) {
    if (l == r) return a[l];
    int t = 31 - __builtin_clz(r ^ l);
    return comb(stor[l][t], stor[r][t]);
  }
  #undef id
  #undef comb
};
```

CaterpillowTree.h

Description: 64-ary set
Usage: bruh
Time: $\mathcal{O}(\log_{64} N)$.

426eac, 89 lines

```
using ull = unsigned long long;
const int depth = 3;
const int sz = 1 << (depth * 6);

struct Tree {
  vt<ull> seg[depth];

  Tree() {
    F0R (i, depth) seg[i].resize(1 << (6 * i));
  }

  void insert(int x) {
    ROF (d, 0, depth) {
      seg[d][x >> 6] |= 1ull << (x & 63);
      x >>= 6;
    }
  }

  void erase(int x) {
    ull b = 0;
    ROF (d, 0, depth) {
      seg[d][x >> 6] &= ~(1ull << (x & 63));
      seg[d][x >> 6] |= b << (x & 63);
      x >>= 6;
      b = bool(seg[d][x]);
    }
  }

  int next(int x) {
    if (x >= sz) return sz;
    x = std::max(x, 0);
    int d = depth - 1;
    while (true) {
      if (ull m = seg[d][x >> 6] >> (x & 63)) {
        x += __builtin_ctzll(m);
        break;
      }
      x = (x >> 6) + 1;
      if (d == 0 || x >= (1 << (6 * d))) return sz;
      d--;
    }
    while (++d < depth) {
      x = (x << 6) + __builtin_ctzll(seg[d][x]);
    }
    return x;
  }

  int prev(int x) {
    if (x < 0) return -1;
    x = std::min(x, sz - 1);
    int d = depth - 1;
    while (true) {
      if (ull m = seg[d][x >> 6] << (63 - (x & 63))) {
        x -= __builtin_clzll(m);
        break;
      }
      x = (x >> 6) - 1;
      if (d == 0 || x == -1) return -1;
      d--;
    }
    while (++d < depth) {
      x = (x << 6) + 63 - __builtin_clzll(seg[d][x]);
    }
    return x;
  }

  int min() {
    if (empty()) return sz;
    int ans = 0;
    F0R (d, depth) {
      ans <<= 6;
      ans += __builtin_ctzll(seg[d][ans >> 6]);
    }
    return ans;
  }

  int max() {
    if (empty()) return -1;
    int ans = 0;
    F0R (d, depth) {
      ans <<= 6;
      ans += 63 - __builtin_clzll(seg[d][ans >> 6]);
    }
    return ans;
  }

  inline bool empty() { return !seg[0][0]; }
  inline int operator[](int i) { return 1 & (seg[depth - 1][i >> 6] >>
      (i & 63)); }
};
```

Treap.h

Description: Treap with too many operations
Time: $\mathcal{O}(\log N)$

817442, 218 lines

```
using K = ll;
random_device rd;
mt19937 mt(rd());

struct Lazy {
  ll v;
  bool inc, rev;
  void operator+=(const Lazy &b) {
    if (b.inc) v += b.v;
    else v = b.v, inc = false;
    rev ^= b.rev;
  }
};

struct Value {
  ll mx, sum;
  void upd(const Lazy &b, int sz) {
    if (!b.inc) mx = sum = 0;
    mx += b.v, sum += b.v * sz;
  }
  Value operator+(const Value &b) const {
    return {max(mx, b.mx), sum + b.sum};
  }
};

const Lazy LID = {0, true, false};
const Value VID = {INF, 0};

using ptr = struct Node*;

struct Node {
  int pri;
  K key;
  ptr l, r;
  int sz;

  Value val, agg;
  Lazy lazy;

  Node(K key, Value val) : key(key), val(val), agg(val) {
    sz = 1;
    pri = mt();
    l = r = 0;
    lazy = LID;
  }

  ~Node() {
    delete l;
    delete r;
  }
};

int sz(ptr n) { return n ? n->sz : 0; }
Value val(ptr n) { return n ? n->val : VID; }
Value agg(ptr n) { return n ? n->agg : VID; }

ptr push(ptr n) {
  if (!n) return n;
  if (n->lazy.rev) swap(n->l, n->r);
  ptr l = n->l, r = n->r;
  n->val.upd(n->lazy, 1);
  n->agg.upd(n->lazy, n->sz);
  if (l) n->l->lazy += n->lazy;
  if (r) n->r->lazy += n->lazy;
  n->lazy = LID;
  return n;
}

ptr pull(ptr n) {
  ptr l = n->l, r = n->r;
  push(l), push(r);
  n->sz = sz(l) + 1 + sz(r);
  n->agg = agg(l) + n->val + agg(r);
  return n;
}

pair<ptr, ptr> split(ptr n, K k) {
  if (!n) return {n, n};
  push(n);
  if (k <= n->key) {
    auto [l, r] = split(n->l, k);
    n->l = r;
    return {l, pull(n)};
  } else {
    auto [l, r] = split(n->r, k);
    n->r = l;
    return {pull(n), r};
  }
}

pair<ptr, ptr> spliti(ptr n, int i) {
  if (!n) return {n, n};
```

```
  push(n);
  if (i <= sz(n->l)) {
    auto [l, r] = spliti(n->l, i);
    n->l = r;
    return {l, pull(n)};
  } else {
    auto [l, r] = spliti(n->r, i - sz(n->l) - 1);
    n->r = l;
    return {pull(n), r};
  }
}

ptr merge(ptr l, ptr r) {
  if (!l || !r) return l ? l : r;
  push(l), push(r);
  ptr t;
  if (l->pri > r->pri) l->r = merge(l->r, r), t = l;
  else r->l = merge(l, r->l), t = r;
  return pull(t);
}

ptr ins(ptr n, K k, Value val) { // insert k
  auto [l, r] = split(n, k);
  return merge(l, merge(new Node(k, val), r));
}

ptr insi(ptr n, int i, K k, Value val) { // insert before i
  auto [l, r] = spliti(n, i);
  return merge(l, merge(new Node(k, val), r));
}

ptr del(ptr n, K k) { // delete k
  auto a = split(n, k), b = spliti(a.s, 1);
  return merge(a.f, b.s);
}

ptr deli(ptr n, int i) {
  auto b = spliti(n, i + 1), a = spliti(b.f, i);
  return merge(a.f, b.s);
}

ptr find(ptr n, K k) {
  push(n);
  if (!n || n->key == k) return n;
  if (k < n->key) return find(n->l, k);
  else return find(n->r, k);
}

ptr findi(ptr n, int i) {
  push(n);
  if (!n || i == sz(n->l)) return n;
  if (i < sz(n->l)) return find(n->l, i);
  else return find(n->r, i);
}

ptr upd(ptr n, K lo, K hi, Lazy nv) {
  if (lo > hi) return n;
  auto [lhs, r] = split(n, hi + 1);
  auto [l, m] = split(lhs, lo);
  m->lazy += nv;
  return merge(l, merge(m, r));
}

ptr updi(ptr n, int lo, int hi, Lazy nv) {
  if (lo > hi) return n;
  auto [lm, r] = spliti(n, hi + 1);
  auto [l, m] = spliti(lm, lo);
  m->lazy += nv;
  return merge(l, merge(m, r));
}

Value query(ptr &n, K lo, K hi)  {
  auto [lm, r] = split(n, hi + 1);
  auto [l, m] = split(lm, lo);
  Value res = agg(m);
  n = merge(l, merge(m, r));
  return res;
}

Value queryi(ptr &n, int lo, int hi) {
  auto [lm, r] = spliti(n, hi + 1);
  auto [l, m] = spliti(lm, lo);
  Value res = agg(m);
  n = merge(l, merge(m, r));
  return res;
}

int mn(ptr n) {
  assert(n);
  push(n);
  if (n->l) return mn(n->l);
  else return n->key;
}

ptr unite(ptr l, ptr r) {
  if (!l || !r) return l ? l : r;
  // l has the smallest key
  if (mn(l) > mn(r)) swap(l, r);
  ptr res = 0;
  while (r) {
    auto [lt, rt] = split(l, mn(r) + 1);
    res = merge(res, lt);
    tie(l, r) = make_pair(r, rt);
  }
  return merge(res, l);
}

void heapify(ptr n) {
  if (!n) return;
  ptr mx = n;
  if (n->l && n->l->pri > mx->pri) mx = n->l;
  if (n->r && n->r->pri > mx->pri) mx = n->r;
  if (mx != n) swap(n->pri, mx->pri), heapify(mx);
}

ptr build(int l, int r, vt<ptr>& ns) {
  if (l > r) return nullptr;
  if (l == r) return ns[l];
  int m = (r + l) / 2;
  ns[m]->l = build(l, m - 1, ns);
  ns[m]->r = build(m + 1, r, ns);
  heapify(ns[m]);
  return pull(ns[m]);
}

Node* tree;
```

Numerical (4)

4.1 Polynomials and recurrences

Polynomial.h

e9feba, 17 lines

```
struct Poly {
  vt<db> a;
  db operator()(double x) const {
    double val = 0;
    for (int i = size(a); i--;) (val *= x) += a[i];
    return val;
  }
  void diff() {
    FOR (i, 1, size(a)) a[i - 1] = i * a[i];
    a.pop_back();
  }
  void divroot(double x0) {
    db b = a.back(), c; a.back() = 0;
    for (int i = size(a) - 1; i--;) c = a[i], a[i] = a[i + 1] * x0 + b
        , b = c;
    a.pop_back();
  }
};
```

PolyRoots.h

Description: Finds the real roots to a polynomial.
Usage: polyRoots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0
Time: $\mathcal{O}\left(n^2 \log(1/\epsilon)\right)$

"Polynomial.h" f1a961, 23 lines

```
vt<db> poly_roots(Poly p, double xmin, double xmax) {
  if (size(p.a) == 2) { return {-p.a[0] / p.a[1]}; }
  vt<db> ret;
  Poly der = p;
  der.diff();
  auto dr = poly_roots(der, xmin, xmax);
  dr.push_back(xmin - 1);
  dr.push_back(xmax + 1);
  sort(all(dr));
  F0R (i, size(dr) - 1) {
    db l = dr[i], h = dr[i + 1];
    bool sign = p(l) > 0;
    if (sign ^ (p(h) > 0)) {
      F0R (it, 60) { // while (h - l > 1e-8)
        double m = (l + h) / 2, f = p(m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.push_back((l + h) / 2);
    }
  }
  return ret;
}
```

PolyInterpolate.h

Description: Given n points (x[i], y[i]), computes an n-1-degree polynomial p that passes through them: $p(x) = a[0] * x^0 + ... + a[n-1] * x^{n-1}$. For numerical precision, pick $x[k] = c * \cos(k/(n-1) * \pi), k = 0 \ldots n-1$.
Time: $\mathcal{O}\left(n^2\right)$

cc9fb9, 13 lines

```
using vd = vt<db>;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  F0R (k, n - 1) FOR (i, k + 1, n)
    y[i] = (y[i] - y[k]) / (x[i] - x[k]);
  double last = 0; temp[0] = 1;
  F0R (k, n) F0R (i, n) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}
```

BerlekampMassey.h

Description: Recovers any n-order linear recurrence relation from the first $2n$ terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size $\leq n$.
Usage: `berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}`
Time: $\mathcal{O}\left(N^2\right)$

"../number-theory/ModPow.h" ce4d42, 20 lines

```
vt<ll> berlekampMassey(vt<ll> s) {
  int n = size(s), L = 0, m = 0;
  vt<ll> C(n), B(n), T;
  C[0] = B[0] = 1;

  ll b = 1;
  F0R (i, n) { ++m;
    ll d = s[i] % mod;
    FOR (j, 1, L + 1) d = (d + C[j] * s[i - j]) % mod;
    if (!d) continue;
    T = C; ll coef = d * mpow(b, mod - 2) % mod;
    FOR (j, m, n) C[j] = (C[j] - coef * B[j - m]) % mod;
    if (2 * L > i) continue;
    L = i + 1 - L; B = T; b = d; m = 0;
  }

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

LinearRecurrence.h

Description: Generates the k'th term of an n-order linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$, given $S[0 \ldots \geq n-1]$ and $tr[0 \ldots n-1]$. Faster than matrix multiplication. Useful together with Berlekamp–Massey.
Usage: `linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number`
Time: $\mathcal{O}\left(n^2 \log k\right)$

602ede, 26 lines

```
using Poly = vt<ll>;
ll linearRec(Poly S, Poly tr, ll k) {
  int n = size(tr);

  auto combine = [&](Poly a, Poly b) {
    Poly res(n * 2 + 1);
    F0R (i, n + 1) F0R (j, n + 1)
      res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
    for (int i = 2 * n; i > n; --i) F0R (j, n)
      res[i - 1 - j] = (res[i - 1 - j] + res[i] * tr[j]) % mod;
    res.resize(n + 1);
    return res;
  };

  Poly pol(n + 1), e(pol);
  pol[0] = e[1] = 1;

  for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
  }

  ll res = 0;
  F0R (i, n) res = (res + pol[i + 1] * S[i]) % mod;
  return res;
}
```

4.2 Optimization

GoldenSectionSearch.h

Description: Finds the argument minimizing the function f in the interval $[a, b]$ assuming f is unimodal on the interval, i.e. has only one local minimum and no local maximum. The maximum error in the result is eps. Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.
Usage: `db func(db x) { return 4+x+.3*x*x; }`
`db xmin = gss(-1000,1000,func);`
Time: $\mathcal{O}\left(\log((b-a)/\epsilon)\right)$

dad647, 14 lines

```
db gss(db a, db b, db (*f)(db)) {
  db r = (sqrt(5)-1)/2, eps = 1e-7;
  db x1 = b - r*(b-a), x2 = a + r*(b-a);
  db f1 = f(x1), f2 = f(x2);
  while (b - a > eps)
    if (f1 < f2) { //change to > to find maximum
      b = x2; x2 = x1; f2 = f1;
      x1 = b - r*(b-a); f1 = f(x1);
    } else {
      a = x1; x1 = x2; f1 = f2;
      x2 = a + r*(b-a); f2 = f(x2);
    }
  return a;
}
```

HillClimbing.h

Description: Poor man's optimization for unimodal functions

94e3f2, 14 lines

```
using P = arrray<db, 2>;

template<class F> pair<db, P> hillClimb(P start, F f) {
  pair<db, P> cur(f(start), start);
  for (db jmp = 1e9; jmp > 1e-20; jmp /= 2) {
    F0R (j, 100) FOR (dx, -1, 2) F0R (dy, -1, 2) {
      P p = cur.second;
      p[0] += dx * jmp;
      p[1] += dy * jmp;
      cur = min(cur, make_pair(f(p), p));
    }
  }
  return cur;
}
```

Simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$. Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
Usage: `vvd A = {{1,-1}, {-1,1}, {-1,-2}};`
`vd b = {1,1,-4}, c = {-1,-1}, x;`
`T val = LPSolver(A, b, c).solve(x);`
Time: $\mathcal{O}\left(NM * \#pivots\right)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}\left(2^n\right)$ in the general case.

4ae05c, 60 lines

```
#define mp make_pair
using T = db;
using vd = vt<db>;
using vvd = vt<vd>;
const db eps = 1e-9;
const int inf = 1e9;

#define ltj(X) if (s == -1 || mp(X[j], N[j]) < mp(X[s], N[s])) s = j
struct LPSolver {
  int m, n;
  vt<int> N, B;
  vvd D;
  LPSolver(const vvd& A, const vd& b, const vd& c) :
    m(size(b)), n(size(c)), N(n + 1), B(m), D(m + 2, vd(n + 2)) {
    F0R(i, m) F0R(j, n) D[i][j] = A[i][j];
    F0R(i, m) B[i] = n+i, D[i][n] = -1, D[i][n + 1] = b[i];
    F0R(j, n) N[j] = j, D[m][j] = -c[j];
    N[n] = -1; D[m + 1][n] = 1;
  }
  void pivot(int r, int s) {
    T inv = 1 / D[r][s];
    F0R(i, m + 2) if (i != r && abs(D[i][s]) > eps) {
      T binv = D[i][s]*inv;
      F0R (j, n + 2) if (j != s) D[i][j] -= D[r][j]*binv;
      D[i][s] = -binv;
    }
    D[r][s] = 1; F0R(j, n + 2) D[r][j] *= inv; // scale r-th row
    swap(B[r],N[s]);
  }
  bool simplex(int phase) {
    int x = m + phase - 1;
    while (1) {
      int s = -1; F0R (j, n + 1) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return 1;
      int r = -1;
      F0R (i, m) {
        if (D[i][s] <= eps) continue;
        if (r == -1 || mp(D[i][n + 1] / D[i][s], B[i])
              < mp(D[r][n + 1] / D[r][s], B[r])) r = i;
      }
      if (r == -1) return 0;
      pivot(r, s);
    }
  }
  T solve(vd &x) {
    int r = 0; FOR (i, 1, m) if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -eps) {
      pivot(r,n);
      assert(simplex(2));
      if (D[m + 1][n + 1] < -eps) return -inf;
      F0R (i, m) if (B[i] == -1) {
        int s = 0; FOR (j, 1, n + 1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    F0R (i, m) if (B[i] < n) x[B[i]] = D[i][n + 1];
    return ok ? D[m][n + 1] : inf;
  }
};
```

4.3 Matrices

Determinant.h

Description: Calculates determinant of a matrix. Destroys the matrix.
Time: $\mathcal{O}\left(N^3\right)$

ef2d92, 15 lines

```
db det(vt<vt<db>>& a) {
  int n = size(a); db res = 1;
  F0R (i, n) {
    int b = i;
    FOR (j, i + 1, n) if (fabs(a[j][i]) > fabs(a[b][i])) b = j;
    if (i != b) swap(a[i], a[b]), res *= -1;
    res *= a[i][i];
    if (res == 0) return 0;
    FOR (j, i + 1, n) {
      double v = a[j][i] / a[i][i];
      if (v != 0) FOR (k, i + 1, n) a[j][k] -= v * a[i][k];
    }
  }
  return res;
}
```

IntDeterminant.h

Description: Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.
Time: $\mathcal{O}\left(N^3\right)$

3313dc, 18 lines

```
const ll mod = 12345;
ll det(vector<vector<ll>>& a) {
  int n = sz(a); ll ans = 1;
  rep(i,0,n) {
    rep(j,i+1,n) {
      while (a[j][i] != 0) { // gcd step
        ll t = a[i][i] / a[j][i];
        if (t) rep(k,i,n)
          a[i][k] = (a[i][k] - a[j][k] * t) % mod;
        swap(a[i], a[j]);
        ans *= -1;
      }
    }
    ans = ans * a[i][i] % mod;
    if (!ans) return 0;
  }
  return (ans + mod) % mod;
}
```

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.
Time: $\mathcal{O}\left(n^2m\right)$

0ff65f, 38 lines

```
using vd = vt<db>;
const double eps = 1e-12;

int solveLinear(vt<vd>& A, vd& b, vd& x) {
  int n = size(A), m = size(x), rank = 0, br, bc;
  if (n) assert(size(A[0]) == m);
  vi col(m); iota(all(col), 0);

  F0R (i, n) {
    double v, bv = 0;
    FOR (r, i, n) FOR (c, i, m)
      if ((v = fabs(A[r][c])) > bv)
        br = r, bc = c, bv = v;
    if (bv <= eps) {
      FOR (j, i, n) if (fabs(b[j]) > eps) return -1;
      break;
    }
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    F0R (j, n) swap(A[j][i], A[j][bc]);
    bv = 1 / A[i][i];
    FOR (j, i + 1, n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      FOR (k, i + 1, m) A[j][k] -= fac * A[i][k];
    }
    rank++;
  }

  x.assign(m, 0);
  for (int i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    F0R (j, i) b[j] -= A[j][i] * b[i];
  }
  return rank; // (multiple solutions if rank < m)
}
```

SolveLinear2.h

Description: To get all uniquely determined values of x back from SolveLinear, make the following changes:

"SolveLinear.h" 77b9bb, 7 lines

```
F0R (j, n) if (j != i) // instead of FOR (j, i + 1, n)
// ... then at the end:
x.assign(m, undefined);
F0R (i, rank) {
  FOR (j, rank, m) if (fabs(A[i][j]) > eps) goto fail;
  x[col[i]] = b[i] / A[i][i];
fail:; }
```

SolveLinearBinary.h

Description: Solves $Ax = b$ over $\mathbb{F}_2$. If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys A and b.
Time: $\mathcal{O}\left(n^2m\right)$

042fea, 34 lines

```
typedef bitset<1000> bs;

int solveLinear(vt<bs>& A, vi& b, bs& x, int m) {
  int n = size(A), rank = 0, br;
  assert(m <= size(x));
  vi col(m); iota(all(col), 0);
  F0R (i, n) {
    FOR (br, i, n) if (A[br].any()) break;
    if (br == n) {
      F0R (j, i, n) if(b[j]) return -1;
      break;
    }
    int bc = (int) A[br]._Find_next(i - 1);
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    F0R (j, n) if (A[j][i] != A[j][bc]) {
      A[j].flip(i); A[j].flip(bc);
    }
    FOR (j, i + 1, n) if (A[j][i]) {
      b[j] ^= b[i];
      A[j] ^= A[i];
    }
    rank++;
  }

  x = bs();
  for (int i = rank; i--;) {
    if (!b[i]) continue;
    x[col[i]] = 1;
    F0R (j, i) b[j] ^= A[j][i];
  }
  return rank; // (multiple solutions if rank < m)
}
```

MatrixInverse.h

Description: Invert matrix A. Returns rank; result is stored in A unless singular (rank $<$ n). Can easily be extended to prime moduli; for prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}\left(n^3\right)$

cb738d, 35 lines

```
int matInv(vt<vt<db>>& A) {
  int n = size(A); vi col(n);
  vt<vt<db>> tmp(n, vt<db>(n));
  F0R (i, n) tmp[i][i] = 1, col[i] = i;

  F0R (i, n) {
    int r = i, c = i;
    FOR (j, i, n) FOR (k, i, n)
      if (fabs(A[j][k]) > fabs(A[r][c]))
        r = j, c = k;
    if (fabs(A[r][c]) < 1e-12) return i;
    A[i].swap(A[r]); tmp[i].swap(tmp[r]);
    F0R (j, n)
      swap(A[j][i], A[j][c]), swap(tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    double v = A[i][i];
    FOR (j, i + 1, n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      FOR (k, i + 1, n) A[j][k] -= f * A[i][k];
      FOR (k, 0, n) tmp[j][k] -= f * tmp[i][k];
    }
    FOR (j, i + 1, n) A[i][j] /= v;
    F0R (j, n) tmp[i][j] /= v;
    A[i][i] = 1;
  }

  for (int i = n - 1; i > 0; --i) F0R (j, i) {
    double v = A[j][i];
    F0R (k, n) tmp[j][k] -= v * tmp[i][k];
  }

  F0R (i, n) F0R (j, n) A[col[i]][col[j]] = tmp[i][j];
  return n;
}
```

4.4 Fourier transforms

FastFourierTransform.h

Description: fft(a) computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. N must be a power of 2. Useful for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, divide by n, reverse(start+1, end), FFT back. Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs). Otherwise, use NTT/FFTMod.
Time: $\mathcal{O}\left(N \log N\right)$ with $N = |A| + |B|$ ($\sim$0.2s for $N = 2^{20}$)

636a47, 36 lines

```
using C = complex<db>;
using vd = vt<db>;

void fft(vt<C> &a) {
  int n = size(a), L = 31 - __builtin_clz(n);
  static vt<complex<long double>> R(2, 1);
  static vt<C> rt(2, 1);  // (^ 10% faster if db)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    FOR (i, k, 2 * k) rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
  }
  vi rev(n);
  F0R (i, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  F0R (i, n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) F0R (j, k) {
      C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)
      a[i + j + k] = a[i + j] - z;
      a[i + j] += z;
    }
}
vd conv(const vd &a, const vd &b) {
  if (a.empty() || b.empty()) return {};
  vd res(size(a) + size(b) - 1);
  int L = 32 - __builtin_clz(size(res)), n = 1 << L;
  vt<C> in(n), out(n);
  copy(all(a), begin(in));
  F0R (i, size(b)) in[i].imag(b[i]);
  fft(in);
  for (C& x : in) x *= x;
  F0R (i, n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
```

```
  F0R (i, size(res)) res[i] = imag(out[i]) / (4 * n);
  return res;
}
```

FastFourierTransformMod.h

Description: Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT) (but seemed +10% on yosupo?)

"FastFourierTransform.h" c4a07e, 22 lines

```
using vl = vt<ll>;
template<int M> vl convMod(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  vl res(size(a) + size(b) - 1);
  int B = 32 - __builtin_clz(size(res)), n = 1 << B, cut = int(sqrt(M)
      );
  vt<C> L(n), R(n), outs(n), outl(n);
  F0R (i, size(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
  F0R (i, size(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
  fft(L), fft(R);
  F0R (i, n) {
    int j = -i & (n - 1);
    outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
    outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
  }
  fft(outl), fft(outs);
  F0R (i, size(res)) {
    ll av = ll(real(outl[i]) + .5), cv = ll(imag(outs[i]) + .5);
    ll bv = ll(imag(outl[i]) + .5) + ll(real(outs[i]) + .5);
    res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
  }
  return res;
}
```

NumberTheoreticTransform.h

Description: ntt(a) computes $\hat{f}(k) = \sum_x a[x] g^{xk}$ for all k, where $g = \text{root}^{(mod-1)/N}$. N must be a power of 2. Useful for convolution modulo specific nice primes of the form $2^a b + 1$, where the convolution result has size at most 2^a. For arbitrary modulo, see FFTMod. `conv(a, b) = c`, where $c[x] = \sum a[i] b[x-i]$. For manual convolution: NTT the inputs, multiply pointwise, divide by n, reverse(start+1, end), NTT back. Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 0.2s$ for $N = 2^{20}$)

"../number-theory/ModPow.h" 8b0fb2, 37 lines

```
const ll root = 62; // = 998244353
// For p < 2^30 there is also e.g. 5 << 25, 7 << 26, 479 << 21
// and 483 << 21 (same root). The last two are > 10^9.

template<class T>
void ntt(vt<T> &a) {
  int n = size(a), L = 31 - __builtin_clz(n);
  static vt<ll> rt(2, 1);
  for (static int k = 2, s = 2; k < n; k *= 2, s++) {
    rt.resize(n);
    ll z[] = {1, mpow(root, mod >> s)};
    FOR (i, k, 2 * k) rt[i] = rt[i / 2] * z[i & 1] % mod;
  }
  vi rev(n);
  FOR (i, 0, n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  FOR (i, 0, n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) FOR (j, 0, k) {
      T z = (ll) rt[j + k] * a[i + j + k] % mod, &ai = a[i + j];
      a[i + j + k] = ai - z + (z > ai ? mod : 0);
      ai += (ai + z >= mod ? z - mod : z);
    }
}

template<class T>
vt<T> conv(const vt<T> &a, const vt<T> &b) {
  if (a.empty() || b.empty()) return {};
  int s = size(a) + size(b) - 1, B = 32 - __builtin_clz(s),
    n = 1 << B;
  int inv = mpow(n, mod - 2);
  vt<T> L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  F0R (i, n) out[-i & (n - 1)] = (ll) L[i] * R[i] % mod * inv % mod;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
```

FastSubsetTransform.h

Description: Transform to a basis with fast convolutions of the form $c[z] = \sum_{z=x \oplus y} a[x] \cdot b[y]$, where $\oplus$ is one of AND, OR, XOR. The size of a must be a power of two. Replace with long longs and do operations under mod if needed.
Time: $\mathcal{O}(N \log N)$

97380b, 17 lines

```
using pi = pair<int, int>;
void FST(vi &a, bool inv) {
  for (int n = size(a), step = 1; step < n; step *= 2) {
    for (int i = 0; i < n; i += 2 * step) FOR (j, i, i + step) {
      int &u = a[j], &v = a[j + step]; tie(u, v) =
        inv ? pi(v - u, u) : pi(v, u + v); // AND
        inv ? pi(v, u - v) : pi(u + v, u); // OR
        pi(u + v, u - v);           // XOR
    }
  }
  if (inv) for (int& x : a) x /= size(a); // XOR only
}
vi conv(vi a, vi b) {
  FST(a, 0); FST(b, 0);
  F0R (i, size(a)) a[i] *= b[i];
  FST(a, 1); return a;
}
```

Number theory (5)

5.1 Modular arithmetic

ModInverse.h

Description: Pre-computation of modular inverses. Assumes LIM $\leq$ mod and that mod is a prime.

e403d4, 3 lines

```
const ll mod = 1000000007, LIM = 200000;
ll* inv = new ll[LIM] - 1; inv[1] = 1;
FOR (i, 2, LIM) inv[i] = mod - (mod / i) * inv[mod % i] % mod;
```

ModPow.h

d63292, 8 lines

```
const ll mod = 1000000007; // faster if const

ll mpow(ll b, ll e) {
  ll ans = 1;
  for (; e; b = b * b % mod, e /= 2)
    if (e & 1) ans = ans * b % mod;
  return ans;
}
```

ModLog.h

Description: Returns the smallest $x > 0$ s.t. $a^x = b \pmod m$, or -1 if no such x exists. modLog(a,1,m) can be used to calculate the order of a.
Time: $\mathcal{O}(\sqrt{m})$

c040b8, 11 lines

```
ll modLog(ll a, ll b, ll m) {
  ll n = (ll) sqrt(m) + 1, e = 1, f = 1, j = 1;
  unordered_map<ll, ll> A;
  while (j <= n && (e = f = e * a % m) != b % m)
    A[e * b % m] = j++;
  if (e == b % m) return j;
  if (__gcd(m, e) == __gcd(m, b))
    rep(i,2,n+2) if (A.count(e = e * f % m))
      return n * i - A[e];
  return -1;
}
```

ModSum.h

Description: Sums of mod'ed arithmetic progressions.
modsum(to, c, k, m) $= \sum_{i=0}^{to-1} (ki + c) \% m$. divsum is similar but for floored division.
Time: $\log(m)$, with a large constant.

5c5bc5, 16 lines

```
typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }

ull divsum(ull to, ull c, ull k, ull m) {
  ull res = k / m * sumsq(to) + c / m * to;
  k %= m; c %= m;
  if (!k) return res;
  ull to2 = (to * k + c) / m;
  return res + (to - 1) * to2 - divsum(to2, m-1 - c, m, k);
}

ll modsum(ull to, ll c, ll k, ll m) {
  c = ((c % m) + m) % m;
  k = ((k % m) + m) % m;
  return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
}
```

ModMulLL.h

Description: Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.
Time: $\mathcal{O}(1)$ for modmul, $\mathcal{O}(\log b)$ for modpow

bb4d60, 11 lines

```
typedef unsigned long long ull;
ull mmul(ull a, ull b, ull M) {
  ll ret = a * b - M * ull(1.L / M * a * b);
  return ret + M * (ret < 0) - M * (ret >= (ll) M);
}
ull mpow(ull b, ull e, ull mod) {
  ull ans = 1;
  for (; e; b = mmul(b, b, mod), e /= 2)
    if (e & 1) ans = mmul(ans, b, mod);
  return ans;
}
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots. Finds x s.t. $x^2 = a \pmod p$ ($-x$ gives the other solution).
Time: $\mathcal{O}(\log^2 p)$ worst case, $\mathcal{O}(\log p)$ for most p

"ModPow.h" 4ba904, 24 lines

```
ll sqrt(ll a, ll p) {
  a %= p; if (a < 0) a += p;
  if (a == 0) return 0;
  assert(mpow(a, (p-1)/2, p) == 1); // else no solution
  if (p % 4 == 3) return mpow(a, (p+1)/4, p);
  // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
  ll s = p - 1, n = 2;
  int r = 0, m;
  while (s % 2 == 0)
    ++r, s /= 2;
  while (mpow(n, (p - 1) / 2, p) != p - 1) ++n;
  ll x = mpow(a, (s + 1) / 2, p);
  ll b = mpow(a, s, p), g = mpow(n, s, p);
```

```
  for (;; r = m) {
    ll t = b;
    for (m = 0; m < r && t != 1; ++m)
      t = t * t % p;
    if (m == 0) return x;
    ll gs = mpow(g, 1LL << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
  }
}
```

5.2 Primality

FastEratosthenes.h

Description: Prime sieve for generating all primes smaller than LIM.
Time: LIM=1e9 ≈ 1.5s

6b2912, 20 lines

```
const int LIM = 1e6;
bitset<LIM> isPrime;
vi eratosthenes() {
  const int S = (int)round(sqrt(LIM)), R = LIM / 2;
  vi pr = {2}, sieve(S+1); pr.reserve(int(LIM/log(LIM)*1.1));
  vector<pii> cp;
  for (int i = 3; i <= S; i += 2) if (!sieve[i]) {
    cp.push_back({i, i * i / 2});
    for (int j = i * i; j <= S; j += 2 * i) sieve[j] = 1;
  }
  for (int L = 1; L <= R; L += S) {
    array<bool, S> block{};
    for (auto &[p, idx] : cp)
      for (int i=idx; i < S+L; idx = (i+=p)) block[i-L] = 1;
    rep(i,0,min(S, R - L))
      if (!block[i]) pr.push_back((L + i) * 2 + 1);
  }
  for (int i : pr) isPrime[i] = 1;
  return pr;
}
```

MillerRabin.h

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger numbers, use Python and extend A randomly.
Time: 7 times the complexity of $a^b \mod c$.

"ModMulLL.h" 60dcd1, 12 lines

```
bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
  ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
    s = __builtin_ctzll(n-1), d = n >> s;
  for (ull a : A) {   // ^ count trailing zeroes
    ull p = modpow(a%n, d, n), i = s;
    while (p != 1 && p != n - 1 && a % n && i--)
      p = modmul(p, p, n);
    if (p != n-1 && i != s) return 0;
  }
  return 1;
}
```

Factor.h

Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).
Time: $\mathcal{O}\left(n^{1/4}\right)$, less for numbers with small factors.

"ModMulLL.h", "MillerRabin.h" d8d98d, 18 lines

```
ull pollard(ull n) {
  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  auto f = [&](ull x) { return modmul(x, x, n) + i; };
  while (t++ % 40 || __gcd(prd, n) == 1) {
    if (x == y) x = ++i, y = f(x);
    if ((q = modmul(prd, max(x,y) - min(x,y), n))) prd = q;
    x = f(x), y = f(f(y));
  }
  return __gcd(prd, n);
}
vector<ull> factor(ull n) {
  if (n == 1) return {};
  if (isPrime(n)) return {n};
  ull x = pollard(n);
  auto l = factor(x), r = factor(n / x);
  l.insert(l.end(), all(r));
  return l;
}
```

5.3 Divisibility

euclid.h

Description: Finds two integers x and y, such that $ax + by = \gcd(a, b)$. If you just need gcd, use the built in __gcd instead. If a and b are coprime, then x is the inverse of $a \pmod b$.

33ba8f, 5 lines

```
ll euclid(ll a, ll b, ll &x, ll &y) {
  if (!b) return x = 1, y = 0, a;
  ll d = euclid(b, a % b, y, x);
  return y -= a/b * x, d;
}
```

CRT.h

Description: Chinese Remainder Theorem.
crt(a, m, b, n) computes x such that $x \equiv a \pmod m$, $x \equiv b \pmod n$. If $|a| < m$ and $|b| < n$, x will obey $0 \le x < \text{lcm}(m, n)$. Assumes $mn < 2^{62}$.
Time: $\log(n)$

"euclid.h" 04d93a, 7 lines

```
ll crt(ll a, ll m, ll b, ll n) {
  if (n > m) swap(a, b), swap(m, n);
  ll x, y, g = euclid(m, n, x, y);
  assert((a - b) % g == 0); // else no solution
  x = (b - a) % n * x % n / g * m + a;
  return x < 0 ? x + m*n/g : x;
}
```

5.3.1 Bézout's identity

For $a \neq, b \neq 0$, then $d = gcd(a, b)$ is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If (x, y) is one solution, then all solutions are given by

$$\left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)}\right), \quad k \in \mathbb{Z}$$

phiFunction.h

Description: *Euler's* ϕ function is defined as $\phi(n) :=$ # of positive integers $\le n$ that are coprime with n. $\phi(1) = 1$, p prime $\Rightarrow \phi(p^k) = (p-1)p^{k-1}$, m, n coprime $\Rightarrow \phi(mn) = \phi(m)\phi(n)$. If $n = p_1^{k_1} p_2^{k_2} ... p_r^{k_r}$ then $\phi(n) = (p_1 - 1)p_1^{k_1-1} ... (p_r - 1)p_r^{k_r-1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$.
$\sum_{d|n} \phi(d) = n$, $\sum_{1 \le k \le n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$
Euler's thm: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod n$.
Fermat's little thm: p prime $\Rightarrow a^{p-1} \equiv 1 \pmod p \ \forall a$.

5bf43c, 8 lines

```
const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
  F0R (i, LIM) phi[i] = i & 1 ? i : i / 2;
  for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
    for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

5.4 Pythagorean Triples

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \ \ b = k \cdot (2mn), \ \ c = k \cdot (m^2 + n^2),$$

with $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

5.5 Estimates

$\sum_{d|n} d = O(n \log \log n)$.

The number of divisors of n is at most around 100 for $n < 5e4$, 250 for $n < 1e6$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

5.6 Mobius Function

$$\mu(n) = \begin{cases} 0 & n \text{ is not square free} \\ 1 & n \text{ has even number of prime factors} \\ -1 & n \text{ has odd number of prime factors} \end{cases}$$

Mobius Inversion:

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu(d) g(n/d)$$

Other useful formulas/forms:

$\sum_{d|n} \mu(d) = [n = 1]$ (very useful)

$g(n) = \sum_{n|d} f(d) \Leftrightarrow f(n) = \sum_{n|d} \mu(d/n) g(d)$

$g(n) = \sum_{1 \le m \le n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum_{1 \le m \le n} \mu(m) g(\lfloor \frac{n}{m} \rfloor)$

Combinatorial (6)

6.1 Permutations

6.1.1 Factorial

n	1	2	3	4	5	6	7	8	9	10
$n!$	1	2	6	24	120	720	5040	40320	362880	3628800
n	11	12	13	14	15	16	17			
$n!$	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14			
n	20	25	30	40	50	100	150	171		
$n!$	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX		

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.
Time: $\mathcal{O}(n)$

044568, 6 lines

```
int permToInt(vi &v) {
  int use = 0, i = 0, r = 0;
  for (int x : v) r = r * ++i + __builtin_popcount(use & -(1 << x)),
    use |= 1 << x;          // (note: minus, not ~!)
  return r;
}
```

6.1.2 Cycles

Let $g_S(n)$ be the number of n-permutations whose cycle lengths all belong to the set S. Then

$$\sum_{n=0}^{\infty} g_S(n)\frac{x^n}{n!} = \exp\left(\sum_{n\in S}\frac{x^n}{n}\right)$$

6.1.3 Derangements

Permutations of a set such that none of the elements appear in their original position.

$$D(n) = (n-1)(D(n-1)+D(n-2)) = nD(n-1)+(-1)^n = \left\lfloor \frac{n!}{e} \right\rceil$$

6.2 Partitions and subsets

6.2.1 Partition function

Number of ways of writing n as a sum of positive integers, disregarding the order of the summands.

$$p(0)=1,\ p(n) = \sum_{k\in\mathbb{Z}\setminus\{0\}} (-1)^{k+1} p(n-k(3k-1)/2)$$

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

n	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	∼2e5	∼2e8

6.2.2 Lucas' Theorem

Let n, m be non-negative integers and p a prime. Write $n = n_k p^k + ... + n_1 p + n_0$ and $m = m_k p^k + ... + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod p$.

6.2.3 Binomials

multinomial.h

Description: Computes $\binom{k_1+\cdots+k_n}{k_1,k_2,\ldots,k_n} = \frac{(\sum k_i)!}{k_1!k_2!\ldots k_n!}$. a0a312, 5 lines

```
ll multinomial(vi& v) {
  ll c = 1, m = v.empty() ? 1 : v[0];
  rep(i,1,sz(v)) rep(j,0,v[i]) c = c * ++m / (j+1);
  return c;
}
```

6.3 General purpose numbers

6.3.1 Bernoulli numbers

EGF of Bernoulli numbers is $B(t) = \frac{t}{e^t-1}$ (FFT-able).
$B[0,\ldots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \ldots]$

Sums of powers:

$$\sum_{i=1}^{n} n^m = \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k} B_k \cdot (n+1)^{m+1-k}$$

Euler-Maclaurin formula for infinite sums:

$$\sum_{i=m}^{\infty} f(i) = \int_m^{\infty} f(x)dx - \sum_{k=1}^{\infty}\frac{B_k}{k!}f^{(k-1)}(m)$$
$$\approx \int_m^{\infty} f(x)dx + \frac{f(m)}{2} - \frac{f'(m)}{12} + \frac{f'''(m)}{720} + O(f^{(5)}(m))$$

6.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

$$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k),\ c(0,0)=1$$
$$\textstyle\sum_{k=0}^{n} c(n,k)x^k = x(x+1)\ldots(x+n-1)$$

$c(8,k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$
$c(n,2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \ldots$

6.3.3 Eulerian numbers

Number of permutations $\pi \in S_n$ in which exactly k elements are greater than the previous element. k j:s s.t. $\pi(j) > \pi(j+1)$, $k+1$ j:s s.t. $\pi(j) \geq j$, k j:s s.t. $\pi(j) > j$.

$$E(n,k) = (n-k)E(n-1,k-1) + (k+1)E(n-1,k)$$

$$E(n,0) = E(n,n-1) = 1$$

$$E(n,k) = \sum_{j=0}^{k}(-1)^j\binom{n+1}{j}(k+1-j)^n$$

6.3.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

$$S(n,k) = S(n-1,k-1) + kS(n-1,k)$$

$$S(n,1) = S(n,n) = 1$$

$$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n$$

6.3.5 Bell numbers

Total number of partitions of n distinct elements. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \ldots$. For p prime,

$$B(p^m+n) \equiv mB(n) + B(n+1) \pmod p$$

6.3.6 Labeled unrooted trees

on n vertices: n^{n-2}
on k existing trees of size n_i: $n_1 n_2 \cdots n_k n^{k-2}$
with degrees d_i: $(n-2)!/((d_1-1)!\cdots(d_n-1)!)$

6.3.7 Catalan numbers

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

$$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2}C_n,\ C_{n+1} = \sum C_i C_{n-i}$$

$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

Graph (7)

7.1 Fundamentals

BellmanFord.h

Description: Calculates shortest paths from s in a graph that might have negative edge weights. Unreachable nodes get dist = inf; nodes reachable through negative-weight cycles get dist = -inf. Assumes $V^2 \max|w_i| < \sim 2^{63}$.
Time: $\mathcal{O}(VE)$ f021ee, 23 lines

```
const ll inf = LLONG_MAX;
struct Ed { int a, b, w, s() { return a < b ? a : -a; }};
struct Node { ll dist = inf; int prev = -1; };

void bellmanFord(vt<Node>& nodes, vt<Ed>& eds, int s) {
  nodes[s].dist = 0;
  sort(all(eds), [](Ed a, Ed b) { return a.s() < b.s(); });

  int lim = size(nodes) / 2 + 2; // /3+100 with shuffled vertices
  F0R (i, lim) for (Ed ed : eds) {
    Node cur = nodes[ed.a], &dest = nodes[ed.b];
    if (abs(cur.dist) == inf) continue;
    ll d = cur.dist + ed.w;
    if (d < dest.dist) {
      dest.prev = ed.a;
      dest.dist = (i < lim-1 ? d : -inf);
    }
  }
  F0R (i, lim) for (Ed e : eds) {
    if (nodes[e.a].dist == -inf)
      nodes[e.b].dist = -inf;
  }
}
```

FloydWarshall.h

Description: Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix m, where $m[i][j] =$ inf if i and j are not adjacent. As output, $m[i][j]$ is set to the shortest distance between i and j, inf if no path, or -inf if the path goes through a negative-weight cycle.
Time: $\mathcal{O}(N^3)$ 35be6b, 10 lines

```
const ll inf = 1ll << 62;
void floydWarshall(vt<vt<ll>>& m) {
  int n = size(m);
  F0R (i, n) m[i][i] = min(m[i][i], 0LL);
  F0R (k, n) F0R (i, n) F0R (j, n)
    if (m[i][k] != inf && m[k][j] != inf)
      m[i][j] = min(m[i][j], max(m[i][k] + m[k][j], -inf));
  F0R (k, n) if (m[k][k] < 0) F0R (i, n) F0R (j, n)
    if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

7.2 Network flow

FordFulkerson.h

Description: Short algo for computing maximum flows with a bounded answer.
Time: $\mathcal{O}(FM)$

86c231, 16 lines

```
const int mx = 2000;
int seen[mx], tim;
unordered_map<int, int> adj[mx];

int flow(int s, int t) {
  auto dfs = [&] (auto &&self, int u) {
    if (u == t) return 1;
    if (exchange(seen[u], tim) == tim) return 0;
    for (auto &[v, c] : adj[u])
      if (c && self(self, v)) return --adj[u][v], ++adj[v][u];
    return 0;
  };
  int flow = 0;
  while (tim++, dfs(dfs, s)) flow++;
  return flow;
}
```

EdmondsKarp.h

Description: Flow algorithm with guaranteed complexity $O(VE^2)$. To get edge flow values, compare capacities before and after, and take the positive values only.

5505a1, 32 lines

```
template<class T> T edmondsKarp(vector<unordered_map<int, T>>&
    adj, int s, int t) {
  assert(s != t);
  T flow = 0;
  vt<int> par(size(adj)), q = par;
  while (1) {
    fill(all(par), -1);
    int ptr = 1;
    q[0] = s, par[s] = 0;
    F0R (i, ptr) {
      int u = q[i];
      for (auto &[v, c] : adj[u]) {
        if (par[v] == -1 && c) {
          par[v] = u, q[ptr++] = v;
          if (v == t) goto out;
        }
      }
    }
    return flow;
  out:
    T inc = numeric_limits<T>::max();
    for (int y = t; y != s; y = par[y])
      inc = min(inc, adj[par[y]][y]);

    flow += inc;
    for (int y = t; y != s; y = par[y]) {
      int p = par[y];
      if ((adj[p][y] -= inc) <= 0) adj[p].erase(y);
      adj[y][p] += inc;
    }
  }
}
```

Dinic.h

Description: Flow algorithm with complexity $O(VE\log U)$ where $U = \max|\text{cap}|$. $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $O(\sqrt{V}E)$ for bipartite matching.

354d03, 47 lines

```
struct Dinic {
  struct Edge {
    int to, rev;
    ll c, oc;
    ll flow() { return max(oc - c, 0LL); } // if you need flows
  };
  vi lvl, ptr, q;
  vt<vt<Edge>> adj;

  void init(int n) {
    lvl = ptr = q = vi(n);
    adj.resize(n);
  }

  void ae(int a, int b, ll c, ll rcap = 0) {
    adj[a].pb({b, size(adj[b]), c, c});
    adj[b].pb({a, size(adj[a]) - 1, rcap, rcap});
  }
  ll dfs(int v, int t, ll f) {
    if (v == t || !f) return f;
    for (int& i = ptr[v]; i < size(adj[v]); i++) {
      auto &[to, rev, c, _] = adj[v][i];
      if (lvl[to] == lvl[v] + 1)
        if (ll p = dfs(to, t, min(f, c))) {
          c -= p, adj[to][rev].c += p;
          return p;
        }
    }
    return 0;
  }
  ll calc(int s, int t) {
    ll flow = 0; q[0] = s;
    F0R (L, 31) do { // 'int L = 30' maybe faster for random data
      lvl = ptr = vi(size(q));
      int qi = 0, qe = lvl[s] = 1;
      while (qi < qe && !lvl[t]) {
        int v = q[qi++];
        for (Edge e : adj[v])
          if (!lvl[e.to] && e.c >> (30 - L))
            q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
      }
      while (ll p = dfs(s, t, LLONG_MAX)) flow += p;
    } while (lvl[t]);
    return flow;
  }
  bool left_of_min_cut(int a) { return lvl[a] != 0; }
};
```

PushRelabel.h

Description: Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$

fc59f0, 66 lines

```
template<typename flow_t = long long>
struct PushRelabel {
  struct Edge {
    int to, rev;
    flow_t f, c;
  };
  vt<vt<Edge>> g;
  vt<flow_t> ec;
  vt<Edge*> cur;
  vt<vt<int>> hs;
  vt<int> h;

  void init(int n) {
    g.resize(n);
    ec.resize(n);
    cur.resize(n);
    hs.resize(2 * n);
    h.resize(n);
  }

  void ae(int s, int t, flow_t cap, flow_t rcap = 0) {
    if (s == t) return;
    g[s].push_back({t, size(g[t]), 0, cap});
    g[t].push_back({s, size(g[s]) - 1, 0, rcap});
  }
  void add_flow(Edge& e, flow_t f) {
    Edge &back = g[e.to][e.rev];
    if (!ec[e.to] && f)
      hs[h[e.to]].push_back(e.to);
    e.f += f; e.c -= f;
    ec[e.to] += f;
    back.f -= f; back.c += f;
    ec[back.to] -= f;
  }
  flow_t calc(int s, int t) {
    int v = size(g);
    h[s] = v;
    ec[t] = 1;
    vt<int> co(2 * v);
    co[0] = v - 1;
    F0R (i, v) cur[i] = g[i].data();
    for(auto &e : g[s]) add_flow(e, e.c);
    if (size(hs[0]))
    for (int hi = 0; hi >= 0;) {
      int u = hs[hi].back();
      hs[hi].pop_back();
      while (ec[u] > 0) // discharge u
        if (cur[u] == g[u].data() + size(g[u])) {
          h[u] = 1e9;
          for (auto &e : g[u])
            if (e.c && h[u] > h[e.to] + 1)
              h[u] = h[e.to] + 1, cur[u] = &e;
          if (++co[h[u]], !--co[hi] && hi < v)
            F0R (i, v)
              if (hi < h[i] && h[i] < v)
                --co[h[i]], h[i] = v + 1;
          hi = h[u];
        } else if (cur[u]->c && h[u] == h[cur[u]->to] + 1)
          add_flow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
      while (hi >= 0 && hs[hi].empty()) --hi;
    }
    return -ec[s];
  }
  bool leftOfMinCut(int a) { return h[a] >= size(g); }
};
```

MinCostMaxFlow.h

Description: Min-cost max-flow. If costs can be negative, call setpi before maxflow, but not that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}(FE\log(V))$ where F is max flow. $\mathcal{O}(VE)$ for setpi.

ccee40, 79 lines

```
#include <bits/extc++.h>

struct MCMF {
  struct edge {
    int from, to, rev;
    ll cap, cost, flow;
  };
  int N;
  vt<vt<edge>> ed;
  vt<int> seen;
  vt<ll> dist, pi;
  vt<edge*> par;

  MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N), par(N) {}
```

```
  void ae(int from, int to, ll cap, ll cost) {
    if (from == to) return;
    ed[from].push_back(edge{ from, to, size(ed[to]),
      cap,cost,  0 });
    ed[to].push_back(edge{ to,  from, size(ed[from]) - 1,
      0, -cost, 0 });
  }

  void path(int s) {
    fill(all(seen), 0);
    fill(all(dist), INF);
    dist[s] = 0; ll di;

    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vt<decltype(q)::point_iterator> its(N);
    q.push({ 0, s });

    while (!q.empty()) {
      s = q.top().second; q.pop();
      seen[s] = 1; di = dist[s] + pi[s];
      for (edge& e : ed[s]) if (!seen[e.to]) {
        ll val = di - pi[e.to] + e.cost;
        if (e.cap - e.flow > 0 && val < dist[e.to]) {
          dist[e.to] = val;
          par[e.to] = &e;
          if (its[e.to] == q.end())
            its[e.to] = q.push({ -dist[e.to], e.to });
          else
            q.modify(its[e.to], { -dist[e.to], e.to });
        }
      }
    }
    FOR (i, N) pi[i] = min(pi[i] + dist[i], INF);
  }

  pair<ll, ll> maxflow(int s, int t) {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = INF;
      for (edge* x = par[t]; x; x = par[x->from])
        fl = min(fl, x->cap - x->flow);

      totflow += fl;
      for (edge* x = par[t]; x; x = par[x->from]) {
        x->flow += fl;
        ed[x->to][x->rev].flow -= fl;
      }
    }
    FOR (i, N) for(edge& e : ed[i]) totcost += e.cost * e.flow;
    return {totflow, totcost/2};
  }

  // If some costs can be negative, call this before maxflow:
  void setpi(int s) { // (otherwise, leave this out)
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--)
      FOR (i, N) if (pi[i] != INF)
        for (edge& e : ed[i]) if (e.cap)
          if ((v = pi[i] + e.cost) < pi[e.to])
            pi[e.to] = v, ch = 1;
    assert(it >= 0); // negative cost cycle
  }
};
```

MinCut.h

Description: After running max-flow, the left side of a min-cut from s to t is given by all vertices reachable from s, only traversing edges with positive residual capacity.

GlobalMinCut.h

Description: Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Time: $\mathcal{O}\left(V^3\right)$

d4d36f, 21 lines

```
pair<int, vt<int>> globalMinCut(vt<vt<int>> mat) {
  pair<int, vt<int>> best = {INT_MAX, {}};
  int n = size(mat);
  vt<vt<int>> co(n);
  F0R (i, n) co[i] = {i};
  FOR(ph, 1, n) {
    vt<int> w = mat[0];
    size_t s = 0, t = 0;
    F0R (it, n - ph) { // O(V^2) -> O(E log V) with prio. queue
      w[t] = INT_MIN;
      s = t, t = max_element(all(w)) - w.begin();
      F0R (i, n) w[i] += mat[t][i];
    }
    best = min(best, {w[t] - mat[t][t], co[t]});
    co[s].insert(co[s].end(), all(co[t]));
    F0R (i, n) mat[s][i] += mat[t][i];
    F0R (i, n) mat[i][s] = mat[s][i];
    mat[0][t] = INT_MIN;
  }
  return best;
}
```

7.3 Matching

hopcroftKarp.h

Description: Fast incremental bipartite matching. Zero-indexed.
Usage: {operator[]} for the pair of right node i, {n} is the size of the rhs, {add(v)} to add adjacency list of node on lhs
Time: $\mathcal{O}\left(\sqrt{V}E\right)$

0821c1, 39 lines

```
struct Matching : vt<int> {
  vt<vt<int>> adj;
  vt<int> rank, low, pos, vis, seen;
  int k{0};
  // n = size of rhs
  Matching(int n) : vt<int>(n, -1), rank(n) {}
  bool add(vt<int> vec) {
    adj.pb(std::move(vec));
    low.pb(0); pos.pb(0); vis.pb(0);
    if (!adj.back().empty()) {
      int i = k;
      nxt:
        seen.clear();
        if (dfs(size(adj)-1, ++k-i)) return 1;
        for (auto &v : seen) for (auto &e : adj[v])
          if (rank[e] < 1e9 && vis[at(e)] < k)
            goto nxt;
        for (auto &v : seen) {
          low[v] = 1e9;
          for (auto &w : adj[v]) rank[w] = 1e9;
        }
      }
      return 0;
    }
    bool dfs(int v, int g) {
      if (vis[v] < k) vis[v] = k, seen.pb(v);
      while (low[v] < g) {
        int e = adj[v][pos[v]];
        if (at(e) != v && low[v] == rank[e]) {
          rank[e]++;
          if (at(e) == -1 || dfs(at(e), rank[e]))
            return at(e) = v, 1;
        } else if (++pos[v] == size(adj[v])) {
          pos[v] = 0; low[v]++;
        }
      }
      return 0;
    }
  };
```

WeightedMatching.h

Description: Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and returns (min cost, match), where L[i] is matched with R[match[i]]. Negate costs for max cost. Requires $N \leq M$.
Time: $\mathcal{O}\left(N^2M\right)$

2e27e7, 31 lines

```
pair<int, vi> hungarian(const vt<vi> &a) {
  if (a.empty()) return {0, {}};
  int n = size(a) + 1, m = size(a[0]) + 1;
  vi u(n), v(m), p(m), ans(n - 1);
  FOR (i, 1, n) {
    p[0] = i;
    int j0 = 0;
    vi dist(m, INT_MAX), pre(m, -1);
    vector<bool> done(m + 1);
    do {
      done[j0] = true;
      int i0 = p[j0], j1, delta = INT_MAX;
      FOR (j, 1, m) if (!done[j]) {
        auto cur = a[i0 - 1][j - 1] - u[i0] - v[j];
        if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
        if (dist[j] < delta) delta = dist[j], j1 = j;
      }
      FOR (j, m) {
        if (done[j]) u[p[j]] += delta, v[j] -= delta;
        else dist[j] -= delta;
      }
      j0 = j1;
    } while (p[j0]);
    while (j0) { // update alternating path
      int j1 = pre[j0];
      p[j0] = p[j1], j0 = j1;
    }
  }
  FOR (j, 1, m) if (p[j]) ans[p[j] - 1] = j - 1;
  return {-v[0], ans}; // min cost
}
```

Blossom.h

Description: Matching for general graphs. 1-indexed!
Time: $\mathcal{O}\left(NM\right)$

0b8afb, 58 lines

```
struct Blossom {
  int n, h, t, cnt;
  vpi edges;
  vi vis, q, mate, col, fa, pre, he;
  void ae(int u, int v) {
    assert(u && v);
    edges.pb({he[u], v}); he[u] = size(edges) - 1;
    edges.pb({he[v], u}); he[v] = size(edges) - 1;
  }
  inline int get(int u) { return fa[u] == u ? u : fa[u] = get(fa[u]);
       }
  void aug(int u, int v) {
    for (int p; u; u = p, v = pre[p])
      p = mate[v], mate[mate[u] = v] = u;
```

```
  }
  void init(int _n) {
    n = _n;
    vis = q = mate = col = fa = pre = he = vi(n + 1);
  }
  int lca(int u, int v) {
    for (cnt++;; u = pre[mate[u]]) {
      if (v) swap(u, v);
      if (vis[u = get(u)] == cnt) return u;
      vis[u] = cnt;
    }
  }
  void blo(int u, int v, int f) {
    for (int p; get(u) != f; v = p, u = pre[p]) {
      p = mate[u]; pre[u] = v; fa[u] = fa[p] = f;
      if (col[p] != 1) col[q[++t] = p] = 1;
    }
  }
  bool bfs(int u) {
    FOR (i, 1, n + 1) col[i] = 0, fa[i] = i;
    h = 0; q[t = 1] = u; col[u] = 1;
    while (h != t) {
      int x = q[++h];
      for (int i = he[x]; i; i = edges[i].f) {
        int y = edges[i].s;
        if (!col[y]) {
          if (!mate[y]) { aug(y, x); return 1; }
          pre[y] = x;
          col[y] = 2;
          col[q[++t] = mate[y]] = 1;
        } else if (col[y] == 1 && get(x) != get(y)) {
          int p = lca(x, y);
          blo(x, y, p);
          blo(y, x, p);
        }
      }
    }
    return 0;
  }
  int solve() {
    int ans = 0;
    FOR (i, 1, n + 1) if (!mate[i]) ans += bfs(i);
    return ans;
  }
};
```

7.4 DFS algorithms

SCC.h

Description: Finds strongly connected components in a directed graph. `comps` is top-sorted.
Time: $\mathcal{O}(E+V)$

ae000b, 30 lines

```
struct SCC {
  int n;
  vt<vt<int>> adj, radj;
  vt<int> todo, seen, comp, comps; // comps is top sorted
  void init(int _n) {
    n = _n;
    adj = radj = vt<vt<int>>(n);
    comp.resize(n, -1);
    seen.resize(n);
  }
  void ae(int u, int v) {
    adj[u].pb(v);
    radj[v].pb(u);
  }
  void dfs(int u) {
    if (seen[u]++) return;
    for (int v : adj[u]) dfs(v);
    todo.pb(u);
  }
  void rdfs(int u, int w) {
    comp[u] = w;
    for (int v : radj[u]) if (comp[v] == -1) rdfs(v, w);
  }
  void gen() {
    F0R (i, n) dfs(i);
    reverse(all(todo));
    for (int u : todo) if (comp[u] == -1)
      rdfs(u, u), comps.pb(u);
  }
};
```

BiconnectedComponents.h

Description: Finds all biconnected components in an undirected graph. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle. Note that degree 0 nodes are not considered components.
Time: $\mathcal{O}(E+V)$

fb417e, 47 lines

```
struct BCC {
  int n, m, t;
  vt<vt<pair<int, int>>> adj;
  vector<pair<int, int>> edges;
  vt<vt<int>> comps; // lists of edges of bcc
  vt<int> tin, stk, is_art, is_bridge;

  void init(int _n, vt<pair<int, int>> &_edges) {
    n = _n;
    edges = _edges;
    m = size(edges);
    adj.resize(n);
    F0R (i, m) {
      auto [u, v] = edges[i];
      adj[u].pb({v, i});
      adj[v].pb({u, i});
    }
    t = 0;
    tin = is_art = vt<int>(n);
    is_bridge.resize(m);
    F0R (u, n) if (!tin[u]) dfs(u, -1);
    // if we include bridges as 2-node bcc
    F0R (i, m) if (is_bridge[i]) comps.pb({i});
  }

  int dfs(int u, int par) {
    int me = tin[u] = ++t, dp = me;
    for (auto [v, ei] : adj[u]) if (ei != par) {
      if (tin[v]) {
        dp = min(dp, tin[v]);
        if (tin[v] < me)
          stk.push_back(ei);
      } else {
        int si = size(stk), up = dfs(v, ei);
        dp = min(dp, up);
        if (up == me) {
          is_art[u] = 1;
          stk.pb(ei);
          comps.pb({si + all(stk)});
          stk.resize(si);
        } else if (up < me) stk.push_back(ei);
        else { is_bridge[ei] = 1; }
      }
    }
    return dp;
  }
};
```

BlockCutTree.h

Description: First, use BiconnectedComponents to locate VERTEX components (including degree 0 nodes). To build a block cut tree, make a bipartite graph: Put all the normal nodes on the left, and make a new node for each bcc on the right. Draw edges from normal nodes to BCC that contain them. Note that the graph may be disconnected.

2sat.h

Description: Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type $(a||b)\&\&(!a||c)\&\&(d||!b)\&\&...$ becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions (`~x`).
Usage: `TwoSat ts(number of boolean variables);`
`ts.either(0, ~3); // Var 0 is true or var 3 is false`
`ts.setValue(2); // Var 2 is true`
`ts.atMostOne({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true`
`ts.solve(); // Returns true iff it is solvable`
`ts.values[0..N-1] holds the assigned values to the vars`
Time: $\mathcal{O}(N+E)$, where N is the number of boolean variables, and E is the number of clauses.

4f0e72, 51 lines

```
struct TwoSAT {
  int n = 0;
  vt<pi> edges;
  void init(int _n) { n = _n; }
  int add() { return n++; }
  void either(int x, int y) { // x | y
    x = max(2 * x, -1 - 2 * x); // ~(2 * x)
    y = max(2 * y, -1 - 2 * y); // ~(2 * y)
    edges.pb({x, y});
  }
  void implies(int x, int y) { either(~x, y); }
  void force(int x) { either(x, x); }
  void exactly_one(int x, int y) {
    either(x, y), either(~x, ~y);
  }
  void tie(int x, int y) {
    implies(x, y), implies(~x, ~y);
  }
  void nand(int x, int y ) { either(~x, ~y); }
  void at_most_one(const vt<int>& li) {
    if (size(li) <= 1) return;
    int cur = ~li[0];
    FOR (i, 2, size(li)) {
      int next = add();
      either(cur, ~li[i]);
      either(cur,next);
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  }
  vt<bool> solve() {
    SCC scc;
    scc.init(2 * n);
    for(auto& e : edges) {
      scc.ae(e.f ^ 1, e.s);
      scc.ae(e.s ^ 1, e.f);
    }
    scc.gen();
    reverse(all(scc.comps)); // reverse topo order
    for (int i = 0; i < 2 * n; i += 2)
      if (scc.comp[i] == scc.comp[i ^ 1]) return {};
    vt<int> tmp(2 * n);
    for (auto i : scc.comps) {
      if (!tmp[i]) tmp[i] = 1, tmp[scc.comp[i ^ 1]] = -1;
    }
    vt<bool> ans(n);
```

```
    F0R (i, n) ans[i] = tmp[scc.comp[2 * i]] == 1;
    return ans;
  }
};
```

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. For edges, push the edge u came from instead of current node. First, the graph (after removing directivity) must be connected. For undirected graphs, a tour exists when all nodes have even degree. For directed graphs, a tour exists when all nodes have equal in and out degree. For trails, the condition is the same as if you added an edge from t -> s.
Time: $\mathcal{O}(V+E)$

378922, 14 lines

```
int n, m;
vt<vt<pair<int, int>>> adj;
vt<int> ret, used;

//
void dfs(int u) {
  while (adj[u].size()) {
    auto [v, ei] = adj[u].back();
    adj[u].pop_back();
    if (used[ei]++) continue;
    dfs(v);
  }
  ret.push_back(u);
}
```

7.5 Heuristics

MaximumClique.h

Description: Quickly finds a maximum clique of a graph (given as symmetric bitset matrix; self-edges not allowed). Can be used to find a maximum independent set by finding a clique of the complement graph.
Time: Runs in about 1s for n=155 and worst case random graphs (p=.90). Runs faster for sparse graphs.

f7c0bc, 49 lines

```
typedef vector<bitset<200>> vb;
struct Maxclique {
  double limit=0.025, pk=0;
  struct Vertex { int i, d=0; };
  typedef vector<Vertex> vv;
  vb e;
  vv V;
  vector<vi> C;
  vi qmax, q, S, old;
  void init(vv& r) {
    for (auto& v : r) v.d = 0;
    for (auto& v : r) for (auto j : r) v.d += e[v.i][j.i];
    sort(all(r), [](auto a, auto b) { return a.d > b.d; });
    int mxD = r[0].d;
    rep(i,0,sz(r)) r[i].d = min(i, mxD) + 1;
  }
  void expand(vv& R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (sz(R)) {
      if (sz(q) + R.back().d <= sz(qmax)) return;
      q.push_back(R.back().i);
      vv T;
      for(auto v:R) if (e[R.back().i][v.i]) T.push_back({v.i});
      if (sz(T)) {
        if (S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1, mnk = max(sz(qmax) - sz(q) + 1, 1);
        C[1].clear(), C[2].clear();
        for (auto v : T) {
          int k = 1;
          auto f = [&](int i) { return e[v.i][i]; };
          while (any_of(all(C[k]), f)) k++;
          if (k > mxk) mxk = k, C[mxk + 1].clear();
          if (k < mnk) T[j++].i = v.i;
          C[k].push_back(v.i);
        }
        if (j > 0) T[j - 1].d = 0;
        rep(k,mnk,mxk + 1) for (int i : C[k])
          T[j].i = i, T[j++].d = k;
        expand(T, lev + 1);
      } else if (sz(q) > sz(qmax)) qmax = q;
      q.pop_back(), R.pop_back();
    }
  }
  vi maxClique() { init(V), expand(V); return qmax; }
  Maxclique(vb conn) : e(conn), C(sz(e)+1), S(sz(C)), old(S) {
    rep(i,0,sz(e)) V.push_back({i});
  }
};
```

MaximumIndependentSet.h

Description: To obtain a maximum independent set of a graph, find a max clique of the complement. If the graph is bipartite, see MinimumVertexCover.

7.6 Trees

LCA.h

Description: Data structure for computing lowest common ancestors in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected.
Time: $\mathcal{O}(N \log N + Q)$

"../data-structures/RMQ.h" bf464a, 20 lines

```
struct LCA {
  int t = 0;
  vi time, path, ret;
  RMQ<int> rmq;

  LCA(vt<vi>& adj) : time(size(adj)), rmq((dfs(0, -1, adj), ret)) {}
  void dfs(int u, int p, vt<vi> &adj) {
    time[u] = t++;
    for (int v : adj[u]) if (v != p) {
      path.push_back(u), ret.push_back(time[u]);
      dfs(v, u, adj);
    }
  }

  int operator()(int u, int v) {
    if (u == v) return u;
    tie(u, v) = minmax(time[u], time[v]);
    return path[rmq.query(u, v)];
  }
};
```

VirtualTree.h

Description: Computes virtual tree. `pos` is inorder dfs time, and returns pairs of `(par, child)`.
Time: $\mathcal{O}(|S| \log |S|)$

"LCA.h" 4ff13b, 14 lines

```
// pos is dfs time
// pairs of {ancestor, child}
vt<pl> virtual_tree(vt<ll>& nodes) {
  auto cmp = [&] (ll u, ll v) { return pos[u] < pos[v]; };
  sort(all(nodes), cmp);
  int sz = size(nodes);
  FOR (i, sz - 1) nodes.pb(lca(nodes[i], nodes[i + 1]));
  sort(all(nodes), cmp);
  nodes.erase(unique(all(nodes)), nodes.end());
  vt<pl> res;
  FOR (i, (int) size(nodes) - 1)
    res.pb({lca(nodes[i], nodes[i + 1]), nodes[i + 1]});
  return res;
}
```

HLD.h

Description: Decomposes a tree into vertex disjoint heavy paths and light edges such that the path from any leaf to the root contains at most log(n) light edges. Code does additive modifications and max queries, but can support commutative segtree modifications/queries on paths and subtrees. Takes as input the full adjacency list. VALS_EDGES being true means that values are stored in the edges, as opposed to the nodes. All values initialized to the segtree default. Root must be 0.
Time: $\mathcal{O}\left((\log N)^2\right)$

"../data-structures/LazySegmentTree.h" 812f05, 69 lines

```
template<bool in_edges> struct HLD {
  int n;
  vt<vt<int>> adj;
  vt<int> par, root, depth, sz, pos;
  int time;
  SegTree tree;
  void ae(int u, int v) {
    adj[u].pb(v);
    adj[v].pb(u);
  }
  void dfs_sz(int u) {
    sz[u] = 1;
    for (int& v : adj[u]) {
      par[v] = u;
      depth[v] = depth[u] + 1;
      adj[v].erase(find(all(adj[v]), u));
      dfs_sz(v);
      sz[u] += sz[v];
      if (sz[v] > sz[adj[u][0]]) swap(v, adj[u][0]);
    }
  }
  void dfs_hld(int u) {
    pos[u] = time++;
    for (int& v : adj[u]) {
      root[v] = (v == adj[u][0] ? root[u] : v);
      dfs_hld(v);
    }
  }
  void init(int _n) {
    n = _n;
    adj.resize(n);
    par = root = depth = sz = pos = vt<int>(n);
  }
  void gen(int r = 0) {
    par[r] = depth[r] = time = 0;
    dfs_sz(r);
    root[r] = r;
    dfs_hld(r);
    tree.init(n);
  }
  int lca(int u, int v) {
    while (root[u] != root[v]) {
      if (depth[root[u]] > depth[root[v]]) swap(u, v);
      v = par[root[v]];
    }
    return depth[u] < depth[v] ? u : v;
  }
  template <class Op>
  void process(int u, int v, Op op) {
    for (;; v = par[root[v]]) {
      if (pos[u] > pos[v]) swap(u, v);
      if (root[u] == root[v]) break;
      op(pos[root[v]], pos[v] + 1);
    }
    op(pos[u] + in_edges, pos[v] + 1);
```

```
  }
  void upd(int u, int v, ll upd) {
    process(u, v, [&] (int l, int r) {
      tree.upd(l, r, upd);
    });
  }
  ll query(int u, int v) {
    ll res = 0;
    process(u, v, [&] (int l, int r) {
      res = res + tree.query(l, r);
    });
    return res;
  }
};
```

7.7 Math

7.7.1 Number of Spanning Trees

Create an $N \times N$ matrix `mat`, and for each edge $a \to b \in G$, do `mat[a][b]-, mat[b][b]++` (and `mat[b][a]-, mat[a][a]++` if G is undirected). Remove the ith row and column and take the determinant; this yields the number of directed spanning trees rooted at i (if G is undirected, remove any row/column).

7.7.2 Erdős–Gallai theorem

A simple graph with node degrees $d_1 \geq \cdots \geq d_n$ exists iff $d_1 + \cdots + d_n$ is even and for every $k = 1 \ldots n$,

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k).$$

Geometry (8)

8.1 Geometric primitives

Point.h

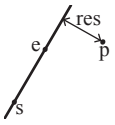
Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

47ec0a, 31 lines

```
// T can be e.g. double or long long. (Avoid int.)
template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x = 0, T y = 0) : x(x), y(y) {}
  bool operator<(P p) const { return tie(x, y) < tie(p.x, p.y); }
  bool operator==(P p) const { return tie(x, y) == tie(p.x, p.y); }
  P operator+(P p) const { return P(x + p.x, y + p.y); }
  P operator-(P p) const { return P(x - p.x, y - p.y); }
  P operator*(T d) const { return P(x * d, y * d); }
  P operator/(T d) const { return P(x / d, y / d); }
  T dot(P p) const { return x * p.x + y * p.y; }
  T cross(P p) const { return x * p.y - y * p.x; }
  T cross(P a, P b) const { return (a - *this).cross(b - *this); }
  T dist2() const { return x * x + y * y; }
  double dist() const { return sqrt((double) dist2()); }
  // angle to x-axis in interval [-pi, pi]
  double angle() const { return atan2(y, x); }
  P unit() const { return *this / dist(); } // makes dist()=1
  P perp() const { return P(-y, x); } // rotates +90 degrees
  P normal() const { return perp().unit(); }
  // returns point rotated 'a' radians ccw around the origin
  P rotate(double a) const {
    return P(x * cos(a) - y * sin(a), x * sin(a) + y * cos(a));
  }
  friend ostream& operator<<(ostream& os, P p) {
    return os << "(" << p.x << "," << p.y << ")";
  }
};
```

lineDistance.h

Description:
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long. Using Point3D will always give a non-negative distance. For Point3D, call .dist on the result of the cross product.

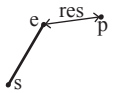


"Point.h" 3b34c4, 4 lines

```
template<class P>
double line_dist(const P& a, const P& b, const P& p) {
  return (double) (b - a).cross(p - a) / (b - a).dist();
}
```

SegmentDistance.h

Description:
Returns the shortest distance between point p and the line segment from point s to e.
Usage: `Point<double> a, b(2,2), p(1,1);`
`bool on_segment = seg_dist(a,b,p) < 1e-10;`

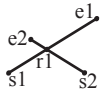


"Point.h" 579797, 5 lines

```
double seg_dist(P& s, P& e, P& p) {
  if (s == e) return (p - s).dist();
  auto d = (e - s).dist2(), t = min(d, max(.0, (p - s).dot(e - s)));
  return ((p - s) * d - (e - s) * t).dist() / d;
}
```

SegmentIntersection.h

Description:
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.
Usage: `vector<P> inter = seg_inter(s1,e1,s2,e2);`
`if (size(inter) == 1)`
`cout << "segments intersect at " << inter[0] << endl;`

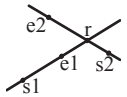


"Point.h", "OnSegment.h" bd6e14, 13 lines

```
template<class P> vt<P> seg_inter(P a, P b, P c, P d) {
  auto oa = c.cross(d, a), ob = c.cross(d, b),
     oc = a.cross(b, c), od = a.cross(b, d);
  // Checks if intersection is single non-endpoint point.
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
    return {(a * ob - b * oa) / (ob - oa)};
  set<P> s;
  if (on_segment(c, d, a)) s.insert(a);
  if (on_segment(c, d, b)) s.insert(b);
  if (on_segment(a, b, c)) s.insert(c);
  if (on_segment(a, b, d)) s.insert(d);
  return {all(s)};
}
```

lineIntersection.h

Description:
If a unique intersection point of the lines going through s1,e1 and s2,e2 exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.
Usage: `auto res = line_inter(s1,e1,s2,e2);`
`if (res.first == 1)`
`cout << "intersection point at " << res.second << endl;`



"Point.h" b0d826, 8 lines

```
template<class P>
pair<int, P> line_inter(P s1, P e1, P s2, P e2) {
  auto d = (e1 - s1).cross(e2 - s2);
  if (d == 0) // if parallel
    return {-(s1.cross(e1, s2) == 0), P(0, 0)};
  auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
  return {1, (s1 * p + e1 * q) / d};
}
```

sideOf.h

Description: Returns where p is as seen from s towards e. 1/0/-1 $\Leftrightarrow$ left/on line/right. If the optional argument eps is given 0 is returned if p is within distance eps from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.
Usage: `bool left = side_of(p1,p2,q)==1;`

"Point.h" 9e71fb, 9 lines

```
template<class P>
int side_of(P s, P e, P p) { return sgn(s.cross(e, p)); }

template<class P>
int side_of(const P& s, const P& e, const P& p, double eps) {
  auto a = (e - s).cross(p - s);
  double l = (e - s).dist() * eps;
  return (a > l) - (a < -l);
}
```

OnSegment.h

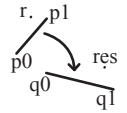
Description: Returns true iff p lies on the line segment from s to e. Use `(seg_dist(s,e,p) < epsilon)` instead when using Point<double>.

"Point.h" 0c1f75, 3 lines

```
template<class P> bool on_segment(P s, P e, P p) {
  return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 0;
}
```

linearTransformation.h

Description:
Apply the linear transformation (translation, rotation and scaling) which takes line p0-p1 to line q0-q1 to point r.



"Point.h" 8e73be, 6 lines

```
typedef Point<double> P;
P linear_transformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) {
  P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
  return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.dist2();
}
```

Angle.h

Description: A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping. Sometimes also represents points or vectors.

Usage: vector<Angle> v = {w[0], w[0].t360() ...}; // sorted
int j = 0; rep(i,0,n) { while (v[j] < v[i].t180()) ++j; }
// sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i

0f0602, 35 lines

```
struct Angle {
  int x, y;
  int t;
  Angle(int x, int y, int t = 0) : x(x), y(y), t(t) {}
  Angle operator-(Angle b) const { return {x - b.x, y - b.y, t}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (half() && x >= 0)}; }
  Angle t180() const { return {-x, -y, t + half()}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare distances
  return make_tuple(a.t, a.half(), a.y * (ll) b.x) <
       make_tuple(b.t, b.half(), a.x * (ll) b.y);
}

// Given two points, this calculates the smallest angle between
// them, i.e., the angle that covers the defined line segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
      make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x * b.x + a.y * b.y, a.x * b.y - a.y * b.x, tu - (b < a)};
}
```

8.2 Circles

CircleIntersection.h

Description: Computes the pair of points at which two circles intersect. Returns false in case of no intersection.

"Point.h" b70280, 11 lines

```
using P = Point<db>;
bool circle_inter(P a, P b, db r1, db r2, pair<P, P> *out) {
  if (a == b) { assert(r1 != r2); return false; }
  P vec = b - a;
  db d2 = vec.dist2(), sum = r1 + r2, dif = r1 - r2,
     p = (d2 + r1 * r1 - r2 * r2) / (d2 * 2), h2 = r1 * r1 - p * p *
          d2;
  if (sum * sum < d2 || dif * dif > d2) return false;
  P mid = a + vec * p, per = vec.perp() * sqrt(fmax(0, h2) / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

CircleTangents.h

Description: Finds the external tangents of two circles, or internal if r2 is negated. Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .first = .second and the tangent line is perpendicular to the line between the centers). .first and .second give the tangency points at circle 1 and 2 respectively. To find the tangents of a circle with a point set r2 to 0.

"Point.h" b0153d, 13 lines

```
template<class P>
vector<pair<P, P>> tangents(P c1, double r1, P c2, double r2) {
  P d = c2 - c1;
  double dr = r1 - r2, d2 = d.dist2(), h2 = d2 - dr * dr;
  if (d2 == 0 || h2 < 0)  return {};
  vector<pair<P, P>> out;
  for (double sign : {-1, 1}) {
    P v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;
    out.push_back({c1 + v * r1, c2 + v * r2});
  }
  if (h2 == 0) out.pop_back();
  return out;
}
```

CircleLine.h

Description: Finds the intersection between a circle and a line. Returns a vector of either 0, 1, or 2 intersection points. P is intended to be Point<double>.

"Point.h" afd5f9, 9 lines

```
template<class P>
vector<P> circle_line(P c, double r, P a, P b) {
  P ab = b - a, p = a + ab * (c - a).dot(ab) / ab.dist2();
  double s = a.cross(b, c), h2 = r * r - s * s / ab.dist2();
  if (h2 < 0) return {};
  if (h2 == 0) return {p};
  P h = ab.unit() * sqrt(h2);
  return {p - h, p + h};
}
```

CirclePolygonIntersection.h

Description: Returns the area of the intersection of a circle with a ccw polygon.
Time: $\mathcal{O}(n)$

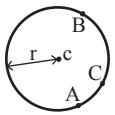
"../../content/geometry/Point.h" e876aa, 19 lines

```
typedef Point<double> P;
#define arg(p, q) atan2(p.cross(q), p.dot(q))
double circlePoly(P c, double r, vector<P> ps) {
  auto tri = [&](P p, P q) {
    auto r2 = r * r / 2;
    P d = q - p;
    auto a = d.dot(p) / d.dist2(), b = (p.dist2() - r * r) / d.dist2()
         ;
    auto det = a * a - b;
    if (det <= 0) return arg(p, q) * r2;
    auto s = max(0., -a - sqrt(det)), t = min(1., -a + sqrt(det));
    if (t < 0 || 1 <= s) return arg(p, q) * r2;
    P u = p + d * s, v = q + d * (t - 1);
    return arg(p, u) * r2 + u.cross(v) / 2 + arg(v, q) * r2;
  };
  auto sum = 0.0;
  F0R (i, size(ps))
    sum += tri(ps[i] - c, ps[(i + 1) % size(ps)] - c);
  return sum;
}
```

circumcircle.h

Description:

The circumcirle of a triangle is the circle intersecting all three vertices. ccRadius returns the radius of the circle going through points A, B and C and ccCenter returns the center of the same circle.



"Point.h" 30a12d, 9 lines

```
typedef Point<double> P;
double cc_radius(const P& A, const P& B, const P& C) {
  return (B - A).dist() * (C - B).dist() * (A - C).dist() /
       abs((B - A).cross(C - A)) / 2;
}
P cc_center(const P& A, const P& B, const P& C) {
  P b = C - A, c = B - A;
  return A + (b * c.dist2() - c * b.dist2()).perp() / b.cross(c) / 2;
}
```

MinimumEnclosingCircle.h

Description: Computes the minimum circle that encloses a set of points.
Time: expected $\mathcal{O}(n)$

"circumcircle.h" 256373, 17 lines

```
pair<P, double> mec(vector<P> ps) {
  shuffle(all(ps), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  F0R (i, size(ps)) if ((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    F0R (j, i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      F0R (k, j) if ((o - ps[k]).dist() > r * EPS) {
        o = cc_center(ps[i], ps[j], ps[k]);
        r = (o - ps[i]).dist();
      }
    }
  }
  return {o, r};
}
```

8.3 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.
Usage: vector<P> v = {P{4,4}, P{1,2}, P{2,1}};
bool in = in_polygon(v, P{3, 3}, false);
Time: $\mathcal{O}(n)$

"Point.h", "OnSegment.h", "SegmentDistance.h" b915a1, 11 lines

```
template<class P>
bool in_polygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = sz(p);
  FOR (i, n) {
    P q = p[(i + 1) % n];
    if (on_segment(p[i], q, a)) return !strict;
    //or: if (segDist(p[i], q, a) <= eps) return !strict;
    cnt ^= ((a.y < p[i].y) - (a.y < q.y)) * a.cross(p[i], q) > 0;
  }
  return cnt;
}
```

PolygonArea.h

Description: Returns twice the signed area of a polygon. Clockwise enumeration gives negative area. Watch out for overflow if using int as T!

"Point.h" c9f086, 6 lines

```
template<class T>
T polygon_area(vector<Point<T>>& v) {
  T a = v.back().cross(v[0]);
  F0R (i, size(v) - 1) a += v[i].cross(v[i + 1]);
  return a;
}
```

PolygonCenter.h

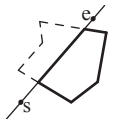
Description: Returns the center of mass for a polygon.
Time: $\mathcal{O}(n)$

"Point.h" ce7a6a, 9 lines

```
typedef Point<db> P;
P polygon_center(const vector<P>& v) {
  P res(0, 0); db a = 0;
```

```
  for (int i = 0, j = size(v) - 1; i < size(v); j = i++) {
    res = res + (v[i] + v[j]) * v[j].cross(v[i]);
    a += v[j].cross(v[i]);
  }
  return res / a / 3;
}
```

PolygonCut.h



Description:
Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.
Usage: vector<P> p = ...;
p = polygonCut(p, P(0,0), P(1,0));

"Point.h" d8e942, 13 lines

```
typedef Point<db> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  F0R (i, size(poly)) {
    P cur = poly[i], prev = i ? poly[i - 1] : poly.back();
    auto a = s.cross(e, cur), b = s.cross(e, prev);
    if ((a < 0) != (b < 0))
      res.push_back(cur + (prev - cur) * (a / (a - b)));
    if (a < 0)
      res.push_back(cur);
  }
  return res;
}
```

PolygonUnion.h

Description: Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)
Time: $\mathcal{O}\left(N^2\right)$, where N is the total number of points

"Point.h", "sideOf.h" b287c2, 33 lines

```
typedef Point<db> P;
double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y; }
double polyUnion(vector<vector<P>>& poly) {
  double ret = 0;
  F0R (i, size(poly)) F0R (v, size(poly[i])) {
    P A = poly[i][v], B = poly[i][(v + 1) % sz(poly[i])];
    vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
    F0R (j, size(poly)) if (i != j) {
      rep(u,0,sz(poly[j])) {
        P C = poly[j][u], D = poly[j][(u + 1) % sz(poly[j])];
        int sc = side_of(A, B, C), sd = side_of(A, B, D);
        if (sc != sd) {
          db sa = C.cross(D, A), sb = C.cross(D, B);
          if (min(sc, sd) < 0)
            segs.emplace_back(sa / (sa - sb), sgn(sc - sd));
        } else if (!sc && !sd && j < i && sgn((B - A).dot(D-C)) > 0){
          segs.emplace_back(rat(C - A, B - A), 1);
          segs.emplace_back(rat(D - A, B - A), -1);
        }
      }
    }
    sort(all(segs));
    for (auto& s : segs) s.first = min(max(s.first, 0.0), 1.0);
    double sum = 0;
    int cnt = segs[0].second;
    FOR (j, 1, size(segs)) {
      if (!cnt) sum += segs[j].first - segs[j - 1].first;
      cnt += segs[j].second;
    }
    ret += A.cross(B) * sum;
  }
  return ret / 2;
}
```

ConvexHull.h

Description:
Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.
Time: $\mathcal{O}\left(n \log n\right)$

"Point.h" 5e9915, 12 lines

```
vt<P> convex_hull(vt<P> pts) {
  if (size(pts) <= 1) return pts;
  sort(all(pts));
  vector<P> h(size(pts)+1);
  int s = 0, t = 0;
  for (int it = 2; it--; s = --t, reverse(all(pts)))
    for (P p : pts) {
      while (t >= s + 2 && h[t - 2].cross(h[t - 1], p) <= 0) t--;
      h[t++] = p;
    }
  return {h.begin(), h.begin() + t - (t == 2 && h[0] == h[1])};
}
```

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).
Time: $\mathcal{O}\left(n\right)$

"Point.h" 8279d4, 12 lines

```
typedef Point<ll> P;
array<P, 2> hull_diameter(vector<P> s) {
  int n = size(s), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {s[0], s[0]}});
  F0R (i, j)
    for (;; j = (j + 1) % n) {
      res = max(res, {(s[i] - s[j]).dist2(), {s[i], s[j]}});
      if ((s[(j + 1) % n] - s[j]).cross(s[i + 1] - s[i]) >= 0)
        break;
    }
  return res.second;
}
```

PointInsideHull.h

Description: Determine whether a point t lies inside a convex hull (CCW order, with no collinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.
Time: $\mathcal{O}\left(\log N\right)$

"Point.h", "sideOf.h", "OnSegment.h" c366a3, 13 lines

```
using P = Point<ll>;
bool in_hull(const vector<P>& l, P p, bool strict = true) {
  int a = 1, b = size(l) - 1, r = !strict;
  if (size(l) < 3) return r && on_segment(l[0], l.back(), p);
  if (side_of(l[0], l[a], l[b]) > 0) swap(a, b);
  if (side_of(l[0], l[a], p) >= r || side_of(l[0], l[b], p) <= -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (side_of(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r;
}
```

LineHullIntersection.h

Description: Line-convex polygon intersection. The polygon must be ccw and have no collinear points. lineHull(line, poly) returns a pair describing the intersection of a line with the polygon: • $(-1,-1)$ if no collision, • $(i,-1)$ if touching the corner i, • (i,i) if along side $(i,i+1)$, • (i,j) if crossing sides $(i,i+1)$ and $(j,j+1)$. In the last case, if a corner i is crossed, this is treated as happening on side $(i,i+1)$. The points are returned in the same order as the line hits the polygon. `extrVertex` returns the point of a hull with the max projection onto a line.
Time: $\mathcal{O}\left(\log n\right)$

"Point.h" bf84fc, 39 lines

```
#define cmp(i,j) sgn(dir.perp().cross(poly[(i) % n] - poly[(j) % n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
template <class P> int extrVertex(vector<P>& poly, P dir) {
  int n = size(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if (extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
    (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
  }
  return lo;
}

#define cmpL(i) sgn(a.cross(poly[i], b))
template <class P>
array<int, 2> lineHull(P a, P b, vector<P>& poly) {
  int endA = extrVertex(poly, (a - b).perp());
  int endB = extrVertex(poly, (b - a).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<int, 2> res;
  F0R (i, 2) {
    int lo = endB, hi = endA, n = size(poly);
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
      (cmpL(m) == cmpL(endB) ? lo : hi) = m;
    }
    res[i] = (lo + !cmpL(hi)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res[0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
      case 0: return {res[0], res[0]};
      case 2: return {res[1], res[1]};
    }
  return res;
}
```

8.4 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.
Time: $\mathcal{O}\left(n \log n\right)$

"Point.h" be22ff, 16 lines

```
pair<P, P> closest(vector<P> v) {
  assert(size(v) > 1);
  set<P> S;
  sort(all(v), [](P a, P b) { return a.y < b.y; });
  pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
  int j = 0;
  for (P p : v) {
    P d{1 + (ll)sqrt(ret.first), 0};
    while (v[j].y <= p.y - d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d);
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).dist2(), {*lo, p}});
    S.insert(p);
  }
  return ret.second;
}
```

kdTree.h

Description: KD-tree (2d)

"Point.h" ad9b75, 53 lines

```
using P = array<int, 2>;
struct Node {
  #define _sq(x) (x) * (x)
  P lo, hi;
  struct Node *lc, *rc;

  ll dist2(const P &a, const P &b) const {
    return 1ll * _sq(a[0] - b[0]) + 1ll * _sq(a[1] - b[1]);
  }

  ll dist2(P &p) {
    #define _loc(i) (p[i] < lo[i] ? lo[i] : (p[i] > hi[i] ? hi[i] : p[
        i]))
    return dist2(p, {_loc(0), _loc(1)});
    // ll res = 0;
    // F0R (i, 2) {
    //   ll tmp = (p[i] < lo[i] ? lo[i] - p[i] : 0) + (hi[i] < p[i] ?
        p[i] - hi[i] : 0);
    //   res += tmp * tmp;
    // }
    // return res;
  }

  template<class ptr>
  Node (ptr l, ptr r, int d) : lc(0), rc(0) {
    lo = {inf, inf}, hi = {-inf, -inf};
    for (ptr p = l; p < r; p++) {
      F0R (i, 2) lo[i] = min(lo[i], (*p)[i]), hi[i] = max(hi[i], (*p)[
          i]);
    }
    if (r - l == 1) return;
    ptr m = l + (r - l) / 2;
    nth_element(l, m, r, [&] (auto a, auto b) { return a[d] < b[d]; })
        ;
    lc = new Node(l, m, d ^ 1);
    rc = new Node(m, r, d ^ 1);
  }

  void search(P p, ll &best) {
    if (lc) { // rc will also exist
      ll dl = lc->dist2(p), dr = rc->dist2(p);
      if (dl > dr) swap(lc, rc), dr = dl;
      lc->search(p, best);
      if (dr < best) rc->search(p, best);
    } else best = min(best, dist2(p, lo));
  }

  // fill pq with k infinities for nearest k points
  void search(P p, priority_queue<ll> &pq) {
    if (lc) {
      ll dl = lc->dist2(p), dr = rc->dist2(p);
      if (dl > dr) swap(lc, rc), dr = dl;
      lc->search(p, pq);
      if (dr < pq.top()) rc->search(p, pq);
    } else pq.push(dist2(p, lo)), pq.pop();
  }
};
```

FastDelaunay.h

Description: Fast Delaunay triangulation. Each circumcircle contains none of the input points. There must be no duplicate points. If all points are on a line, no triangles will be returned. Should work for doubles as well, though there may be precision issues in 'circ'. Returns triangles in order {t[0][0], t[0][1], t[0][2], t[1][0], ...}, all counter-clockwise.

Time: $\mathcal{O}(n \log n)$

"Point.h" eefdf5, 88 lines

```
typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other point

struct Quad {
  Q rot, o; P p = arb; bool mark;
  P& F() { return r()->p; }
  Q& r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
} *H;

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
    B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B > 0;
}
Q makeEdge(P orig, P dest) {
  Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}}};
  H = r->o; r->r()->r() = r;
  rep(i,0,4) r = r->rot, r->p = arb, r->o = i & 1 ? r : r->r();
  r->p = orig; r->F() = dest;
  return r;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

pair<Q,Q> rec(const vector<P>& s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back());
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
  }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = sz(s) / 2;
  tie(ra, A) = rec({all(s) - half});
  tie(B, rb) = rec({sz(s) - half + all(s)});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e->o = H; H = e; e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}

vector<P> triangulate(vector<P> pts) {
  sort(all(pts));  assert(unique(all(pts)) == pts.end());
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p); \
  q.push_back(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

8.5 3D

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

67c08d, 8 lines

```
db sphericalDistance(db f1, db t1,
    db f2, db t2, db radius) {
  db dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  db dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  db dz = cos(t2) - cos(t1);
  db d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}
```

Strings (9)

KMP.h

Description: pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.

Time: $\mathcal{O}(n)$

d4375c, 16 lines

```
vi pi(const string& s) {
  vi p(sz(s));
  rep(i,1,sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

vi match(const string& s, const string& pat) {
  vi p = pi(pat + '\0' + s), res;
  rep(i,sz(p)-sz(s),sz(p))
    if (p[i] == sz(pat)) res.push_back(i - 2 * sz(pat));
  return res;
}
```

Zfunc.h

Description: z[i] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)

Time: $\mathcal{O}(n)$

98e47b, 12 lines

```
vi Z(const string& S) {
```

```
  vi z(size(S));
  int l = -1, r = -1;
  FOR (i, 1, size(S)) {
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    while (i + z[i] < size(S) && S[i + z[i]] == S[z[i]])
      z[i]++;
    if (i + z[i] > r)
      l = i, r = i + z[i];
  }
  return z;
}
```

Manacher.h

Description: For each position in a string, computes p[0][i] = half length of longest even palindrome around pos i, p[1][i] = longest odd (half rounded down).
Time: $\mathcal{O}(N)$

1c24a1, 13 lines

```
array<vi, 2> manacher(const string &s) {
  int n = size(s);
  array<vi, 2> p = {vi(n + 1), vi(n)};
  F0R (z, 2) for (int i = 0, l = 0, r = 0; i < n; i++) {
    int t = r - i + !z;
    if (i < r) p[z][i] = min(t, p[z][l + t]);
    int L = i - p[z][i], R = i + p[z][i] - !z;
    while (L >= 1 && R + 1 < n && s[L - 1] == s[R + 1])
      p[z][i]++, L--, R++;
    if (R > r) l = L, r = R;
  }
  return p;
}
```

MinRotation.h

Description: Finds the lexicographically smallest rotation of a string.
Usage: `rotate(v.begin(), v.begin() + minRotation(v), v.end());`
Time: $\mathcal{O}(N)$

9bd725, 8 lines

```
int minRotation(string s) {
  int a = 0, N = size(s); s += s;
  F0R (b, N) F0R (k, N) {
    if (a + k == b || s[a + k] < s[b + k]) { b += max(0, k - 1); break
        ; }
    if (s[a + k] > s[b + k]) { a = b; break; }
  }
  return a;
}
```

SuffixArray.h

Description: Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. The input string must not contain any nul chars.
Time: $\mathcal{O}(N \log N)$

a982c3, 22 lines

```
struct SuffixArray {
  vi sa, lcp;
  SuffixArray(string s, int lim = 256) { // or vector<int>
    s.push_back(0); int n = size(s), k = 0, a, b;
    vi x(all(s)), y(n), ws(max(n, lim));
    sa = lcp = y, iota(all(sa), 0);
    for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
      p = j, iota(all(y), n - j);
      F0R (i, n) if (sa[i] >= j) y[p++] = sa[i] - j;
      fill(all(ws), 0);
      F0R (i, n) ws[x[i]]++;
      FOR (i, 1, lim) ws[i] += ws[i - 1];
      for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      FOR (i, 1, n) a = sa[i - 1], b = sa[i], x[b] =
        (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
    }
    for (int i = 0, j; i < n - 1; lcp[x[i++]] = k)
      for (k && k--, j = sa[x[i] - 1];
          s[i + k] == s[j + k]; k++);
  }
};
```

SuffixTree.h

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).
Time: $\mathcal{O}(26N)$

aae0b8, 50 lines

```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur position
  int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;

  void ukkadd(int i, int c) { suff:
    if (r[v]<=q) {
      if (t[v][c]==-1) { t[v][c]=m;  l[m]=i;
        p[m++]=v; v=s[v]; q=r[v];  goto suff; }
      v=t[v][c]; q=l[v];
    }
    if (q==-1 || c==toi(a[q])) q++; else {
      l[m+1]=i;  p[m+1]=m;  l[m]=l[v];  r[m]=q;
      p[m]=p[v];  t[m][c]=m+1;  t[m][toi(a[q])]=v;
      l[v]=q;  p[v]=m;  t[p[m]][toi(a[l[m]])]=m;
      v=s[p[m]];  q=l[m];
      while (q<r[m]) { v=t[v][toi(a[q])];  q+=r[v]-l[v]; }
      if (q==r[m])  s[m]=v;  else s[m]=m+2;
      q=r[v]-(q-r[m]);  m+=2;  goto suff;
    }
  }

  SuffixTree(string a) : a(a) {
    fill(r,r+N,sz(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALPHA,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] = 0;
    rep(i,0,sz(a)) ukkadd(i, toi(a[i]));
  }

  // example: find longest common substring (uses ALPHA = 28)
  pii best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node]) : 0;
    rep(c,0,ALPHA) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2));
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
  }
};
```

Hashing.h

Description: Self-explanatory methods for string hashing. Skip the stuff that starts with r if you don't care about reverse hashes etc.

04ecd7, 57 lines

```
// Arithmetic mod 2^64-1. 2x slower than mod 2^64 and more
// code, but works on evil test data (e.g. Thue-Morse, where
// ABBA... and BAAB... of length 2^10 hash the same mod 2^64).
// "typedef ull H;" instead if you think test data is random,
// or work mod 10^9+7 if the Birthday paradox is not a problem.
struct H {
  ull x; H(ull x = 0) : x(x) {}
  H operator+(H o) { return x + o.x + (x + o.x < x); }
  H operator-(H o) { return *this + ~o.x; }
  H operator*(H o) { auto m = (__uint128_t) x * o.x;
    return H((ull) m) + (ull)(m >> 64); }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get() == o.get(); }
  bool operator<(H o) const { return get() < o.get(); }
};

static const H C = (ll) 1e11 + 3; // (order ~ 3e9; random also ok)

struct HashInterval {
  vt<H> ha, pw, rha;
  template<class T>
  HashInterval(T& str) : ha(size(str) + 1), pw(ha), rha(ha) {
    pw[0] = 1;
    FOR (i, size(str)) {
      ha[i + 1] = ha[i] * C + str[i] + 1;
      pw[i + 1] = pw[i] * C;
    }
    ROF (i, size(str)) rha[i] = rha[i + 1] * C + str[i] + 1;
  }
  H hash_interval(int a, int b) { // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
  H rhash_interval(int a, int b) { // hash [a, b) from right to left
    return rha[a] - rha[b] * pw[b - a];
  }
};

// get all hashes of length <len>
template<class T>
vector<H> get_hashes(T& str, int length) {
  if (size(str) < length) return {};
  H h = 0, pw = 1;
  FOR (i, length) h = h * C + str[i] + 1, pw = pw * C;
  vector<H> ret = {h};
  FOR (i, length, size(str)) {
    ret.push_back(h = h * C + str[i] + 1
      - pw * (str[i - length] + 1));
  }
  return ret;
}

template<class T>
H hash_string(T& s) {
  H h = 1;
  for (auto c : s) h = h * C + c + 1;
  return h;
}
```

AhoCorasick.h

Description: Aho-Corasick automaton, used for multiple pattern matching. Initialize with AhoCorasick ac(patterns); the automaton start node will be at index 0. find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll(−, word) finds all words (up to $N\sqrt{N}$ many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. To find the longest words that start at each position, reverse all input. For large alphabets, split each symbol into chunks, with sentinel bits for symbol boundaries.
Time: construction takes $\mathcal{O}(26N)$, where N = sum of length of patterns. find(x) is $\mathcal{O}(N)$, where N = length of x. findAll is $\mathcal{O}(NM)$. f35677, 66 lines

```
struct AhoCorasick {
  enum {alpha = 26, first = 'A'}; // change this!
  struct Node {
    // (nmatches is optional)
    int back, next[alpha], start = -1, end = -1, nmatches = 0;
    Node(int v) { memset(next, v, sizeof(next)); }
  };
  vector<Node> N;
  vi backp;
  void insert(string& s, int j) {
    assert(!s.empty());
    int n = 0;
    for (char c : s) {
      int& m = N[n].next[c - first];
      if (m == -1) { n = m = sz(N); N.emplace_back(-1); }
      else n = m;
    }
    if (N[n].end == -1) N[n].start = j;
    backp.push_back(N[n].end);
    N[n].end = j;
    N[n].nmatches++;
  }
  AhoCorasick(vector<string>& pat) : N(1, -1) {
    rep(i,0,sz(pat)) insert(pat[i], i);
    N[0].back = sz(N);
    N.emplace_back(0);

    queue<int> q;
    for (q.push(0); !q.empty(); q.pop()) {
      int n = q.front(), prev = N[n].back;
      rep(i,0,alpha) {
        int &ed = N[n].next[i], y = N[prev].next[i];
        if (ed == -1) ed = y;
        else {
          N[ed].back = y;
          (N[ed].end == -1 ? N[ed].end : backp[N[ed].start])
            = N[y].end;
          N[ed].nmatches += N[y].nmatches;
          q.push(ed);
        }
      }
    }
  }
  vi find(string word) {
    int n = 0;
    vi res; // ll count = 0;
    for (char c : word) {
      n = N[n].next[c - first];
      res.push_back(N[n].end);
      // count += N[n].nmatches;
    }
    return res;
  }
  vector<vi> findAll(vector<string>& pat, string word) {
    vi r = find(word);
    vector<vi> res(sz(word));
    rep(i,0,sz(word)) {
      int ind = r[i];
      while (ind != -1) {
        res[i - sz(pat[ind]) + 1].push_back(ind);
        ind = backp[ind];
      }
    }
    return res;
  }
};
```

Various (10)

10.1 Intervals

IntervalContainer.h

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).
Time: $\mathcal{O}(\log N)$ edce47, 23 lines

```
set<pii>::iterator addInterval(set<pii>& is, int L, int R) {
  if (L == R) return is.end();
  auto it = is.lower_bound({L, R}), before = it;
  while (it != is.end() && it->first <= R) {
    R = max(R, it->second);
    before = it = is.erase(it);
  }
  if (it != is.begin() && (--it)->second >= L) {
    L = min(L, it->first);
    R = max(R, it->second);
    is.erase(it);
  }
  return is.insert(before, {L,R});
}

void removeInterval(set<pii>& is, int L, int R) {
  if (L == R) return;
  auto it = addInterval(is, L, R);
  auto r2 = it->second;
  if (it->first == L) is.erase(it);
  else (int&)it->second = L;
  if (R != r2) is.emplace(R, r2);
}
```

IntervalCover.h

Description: Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add `|| R.empty()`. Returns empty set on failure (or if G is empty).
Time: $\mathcal{O}(N \log N)$ 9e9d8d, 19 lines

```
template<class T>
vi cover(pair<T, T> G, vector<pair<T, T>> I) {
  vi S(sz(I)), R;
  iota(all(S), 0);
  sort(all(S), [&](int a, int b) { return I[a] < I[b]; });
  T cur = G.first;
  int at = 0;
  while (cur < G.second) { // (A)
    pair<T, int> mx = make_pair(cur, -1);
    while (at < sz(I) && I[S[at]].first <= cur) {
      mx = max(mx, make_pair(I[S[at]].second, S[at]));
      at++;
    }
    if (mx.second == -1) return {};
    cur = mx.first;
    R.push_back(mx.second);
  }
  return R;
}
```

ConstantIntervals.h

Description: Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval.
Usage: `constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});`
Time: $\mathcal{O}\left(k \log \frac{n}{k}\right)$ 753a4c, 19 lines

```
template<class F, class G, class T>
void rec(int from, int to, F& f, G& g, int& i, T& p, T q) {
  if (p == q) return;
  if (from == to) {
    g(i, to, p);
    i = to; p = q;
  } else {
    int mid = (from + to) >> 1;
    rec(from, mid, f, g, i, p, f(mid));
    rec(mid+1, to, f, g, i, p, q);
  }
}
template<class F, class G>
void constantIntervals(int from, int to, F f, G g) {
  if (to <= from) return;
  int i = from; auto p = f(i), q = f(to-1);
  rec(from, to-1, f, g, i, p, q);
  g(i, to, q);
}
```

10.2 Misc. algorithms

TernarySearch.h

Description: Find the smallest i in $[a, b]$ that maximizes $f(i)$, assuming that $f(a) < \ldots < f(i) \geq \cdots \geq f(b)$. To reverse which of the sides allows non-strict inequalities, change the $<$ marked with (A) to $<=$, and reverse the loop at (B). To minimize f, change it to $>$, also at (B).
Usage: `int ind = ternSearch(0,n-1,[&](int i){return a[i];});`
Time: $\mathcal{O}(\log(b-a))$ 9155b4, 11 lines

```
template<class F>
int ternSearch(int a, int b, F f) {
  assert(a <= b);
  while (b - a >= 5) {
    int mid = (a + b) / 2;
    if (f(mid) < f(mid+1)) a = mid; // (A)
    else b = mid+1;
  }
  rep(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
  return a;
}
```

LIS.h

Description: Compute indices for the longest increasing subsequence.
Time: $\mathcal{O}(N \log N)$ 2932a0, 17 lines

```
template<class I> vi lis(const vector<I>& S) {
  if (S.empty()) return {};
  vi prev(sz(S));
  typedef pair<I, int> p;
  vector<p> res;
  rep(i,0,sz(S)) {
    // change 0 -> i for longest non-decreasing subsequence
    auto it = lower_bound(all(res), p{S[i], 0});
    if (it == res.end()) res.emplace_back(), it = res.end()-1;
    *it = {S[i], i};
    prev[i] = it == res.begin() ? 0 : (it-1)->second;
  }
  int L = sz(res), cur = res.back().second;
  vi ans(L);
```

```
  while (L--) ans[L] = cur, cur = prev[cur];
  return ans;
}
```

FastKnapsack.h

Description: Given N non-negative integer weights w and a non-negative target t, computes the maximum S <= t such that S is the sum of some subset of the weights.
Time: $\mathcal{O}(N \max(w_i))$

b20ccc, 16 lines

```
int knapsack(vi w, int t) {
  int a = 0, b = 0, x;
  while (b < sz(w) && a + w[b] <= t) a += w[b++];
  if (b == sz(w)) return a;
  int m = *max_element(all(w));
  vi u, v(2*m, -1);
  v[a+m-t] = b;
  rep(i,b,sz(w)) {
    u = v;
    rep(x,0,m) v[x+w[i]] = max(v[x+w[i]], u[x]);
    for (x = 2*m; --x > m;) rep(j, max(0,u[x]), v[x])
      v[x-w[j]] = max(v[x-w[j]], j);
  }
  for (a = t; v[a+m-t] < 0; a--) ;
  return a;
}
```

10.3 Dynamic programming

KnuthDP.h

Description: When doing DP on intervals: $a[i][j] = \min_{i<k<j}(a[i][k] + a[k][j]) + f(i,j)$, where the (minimal) optimal k increases with both i and j, one can solve intervals in increasing order of length, and search $k = p[i][j]$ for $a[i][j]$ only between $p[i][j-1]$ and $p[i+1][j]$. This is known as Knuth DP. Sufficient criteria for this are if $f(b,c) \leq f(a,d)$ and $f(a,c) + f(b,d) \leq f(a,d) + f(b,c)$ for all $a \leq b \leq c \leq d$. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.
Time: $\mathcal{O}(N^2)$

DivideAndConquerDP.h

Description: Given $a[i] = \min_{lo(i)\leq k<hi(i)}(f(i,k))$ where the (minimal) optimal k increases with i, computes $a[i]$ for $i = L..R-1$.
Time: $\mathcal{O}((N + (hi - lo))\log N)$

d38d2b, 18 lines

```
struct DP { // Modify at will:
  int lo(int ind) { return 0; }
  int hi(int ind) { return ind; }
  ll f(int ind, int k) { return dp[ind][k]; }
  void store(int ind, int k, ll v) { res[ind] = pii(k, v); }

  void rec(int L, int R, int LO, int HI) {
    if (L >= R) return;
    int mid = (L + R) >> 1;
    pair<ll, int> best(LLONG_MAX, LO);
    rep(k, max(LO,lo(mid)), min(HI,hi(mid)))
      best = min(best, make_pair(f(mid, k), k));
    store(mid, best.second, best.first);
    rec(L, mid, LO, best.second+1);
    rec(mid+1, R, best.second, HI);
  }
  void solve(int L, int R) { rec(L, R, INT_MIN, INT_MAX); }
};
```

10.4 Debugging tricks

- `signal(SIGSEGV, [](int) { _Exit(0); });` converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions). `_GLIBCXX_DEBUG` failures generate SIGABRT (or SIGSEGV on gcc 5.4.0 apparently).
- `feenableexcept(29);` kills the program on NaNs (1), 0-divs (4), infinities (8) and denormals (16).

10.5 Optimization tricks

`__builtin_ia32_ldmxcsr(40896);` disables denormals (which make floats 20x slower near their minimum value).

10.5.1 Bit hacks

- `x & -x` is the least bit in `x`.
- `for (int x = m; x; ) { --x &= m; ... }` loops over all subset masks of `m` (except `m` itself).
- `c = x&-x, r = x+c; (((r^x) >> 2)/c) | r` is the next number after `x` with the same number of bits set.
- `rep(b,0,K) rep(i,0,(1 << K)) if (i & 1 << b) D[i] += D[i^(1 << b)];` computes all sums of subsets.

10.5.2 Pragmas

- `#pragma GCC optimize ("Ofast")` will make GCC auto-vectorize loops and optimizes floating points better.
- `#pragma GCC target ("avx2")` can double performance of vectorized code, but causes crashes on old machines.
- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).

FastMod.h

Description: Compute $a\%b$ about 5 times faster than usual, where b is constant but not known at compile time. Returns a value congruent to $a \pmod b$ in the range $[0, 2b)$.

751a02, 8 lines

```
typedef unsigned long long ull;
struct FastMod {
  ull b, m;
  FastMod(ull b) : b(b), m(-1ULL / b) {}
  ull reduce(ull a) { // a % b + (0 or b)
    return a - (ull)((__uint128_t(m) * a) >> 64) * b;
  }
};
```

FastInput.h

Description: Read an integer from stdin. Usage requires your program to pipe in input from file.
Usage: ./a.out < input.txt
Time: About 5x as fast as cin/scanf.

7b3c70, 17 lines

```
inline char gc() { // like getchar()
  static char buf[1 << 16];
  static size_t bc, be;
  if (bc >= be) {
    buf[0] = 0, bc = 0;
    be = fread(buf, 1, sizeof(buf), stdin);
  }
  return buf[bc++]; // returns 0 on EOF
}

int readInt() {
  int a, c;
  while ((a = gc()) < 40);
  if (a == '-') return -readInt();
  while ((c = gc()) >= 48) a = a * 10 + c - 480;
  return a - 48;
}
```

BumpAllocator.h

Description: When you need to dynamically allocate many objects and don't care about freeing them. "new X" otherwise has an overhead of something like 0.05us + 16 bytes per allocation.

745db2, 8 lines

```
// Either globally or in a single class:
static char buf[450 << 20];
void* operator new(size_t s) {
  static size_t i = sizeof buf;
  assert(s < i);
  return (void*)&buf[i -= s];
}
void operator delete(void*) {}
```

SmallPtr.h

Description: A 32-bit pointer that points into BumpAllocator memory.

"BumpAllocator.h" 2dd6c9, 10 lines

```
template<class T> struct ptr {
  unsigned ind;
  ptr(T* p = 0) : ind(p ? unsigned((char*)p - buf) : 0) {
    assert(ind < sizeof buf);
  }
  T& operator*() const { return *(T*)(buf + ind); }
  T* operator->() const { return &**this; }
  T& operator[](int a) const { return (&**this)[a]; }
  explicit operator bool() const { return ind; }
};
```

BumpAllocatorSTL.h

Description: BumpAllocator for STL containers.
Usage: vector<vector<int, small<int>>> ed(N);

bb66d4, 14 lines

```
char buf[450 << 20] alignas(16);
size_t buf_ind = sizeof buf;

template<class T> struct small {
  typedef T value_type;
  small() {}
  template<class U> small(const U&) {}
  T* allocate(size_t n) {
    buf_ind -= n * sizeof(T);
    buf_ind &= 0 - alignof(T);
    return (T*)(buf + buf_ind);
  }
  void deallocate(T*, size_t) {}
};
```

SIMD.h

Description: Cheat sheet of SSE/AVX intrinsics, for doing arithmetic on several numbers at once. Can provide a constant factor improvement of about 4, orthogonal to loop unrolling. Operations follow the pattern "`_mm(256)?_name_(si(128|256)|epi(8|16|32|64)|pd|ps)`". Not all are described here; grep for `_mm_` in `/usr/lib/gcc/*/4.9/include/` for more. If AVX is unsupported, try 128-bit operations, "emmintrin.h" and #define `__SSE__` and `__MMX__` before including it. For aligned memory use `_mm_malloc(size, 32)` or `int buf[N] alignas(32)`, but prefer loadu/storeu.

551b82, 43 lines

```
#pragma GCC target ("avx2") // or sse4.1
#include "immintrin.h"

typedef __m256i mi;
#define L(x) _mm256_loadu_si256((mi*)&(x))

// High-level/specific methods:
// load(u)?_si256, store(u)?_si256, setzero_si256, _mm_malloc
// blendv_(epi8|ps|pd) (z?y:x), movemask_epi8 (hibits of bytes)
// i32gather_epi32(addr, x, 4): map addr[] over 32-b parts of x
// sad_epu8: sum of absolute differences of u8, outputs 4xi64
// maddubs_epi16: dot product of unsigned i7's, outputs 16xi15
// madd_epi16: dot product of signed i16's, outputs 8xi32
// extractf128_si256(, i) (256->128), cvtsi128_si32 (128->lo32)
// permute2f128_si256(x,x,1) swaps 128-bit lanes
// shuffle_epi32(x, 3*64+2*16+1*4+0) == x for each lane
// shuffle_epi8(x, y) takes a vector instead of an imm

// Methods that work with most data types (append e.g. _epi32):
// set1, blend (i8?x:y), add, adds (sat.), mullo, sub, and/or,
// andnot, abs, min, max, sign(1,x), cmp(gt|eq), unpack(lo|hi)

int sumi32(mi m) { union {int v[8]; mi m;} u; u.m = m;
  int ret = 0; rep(i,0,8) ret += u.v[i]; return ret; }
mi zero() { return _mm256_setzero_si256(); }
mi one() { return _mm256_set1_epi32(-1); }
bool all_zero(mi m) { return _mm256_testz_si256(m, m); }
bool all_one(mi m) { return _mm256_testc_si256(m, one()); }

ll example_filteredDotProduct(int n, short* a, short* b) {
  int i = 0; ll r = 0;
  mi zero = _mm256_setzero_si256(), acc = zero;
  while (i + 16 <= n) {
    mi va = L(a[i]), vb = L(b[i]); i += 16;
    va = _mm256_and_si256(_mm256_cmpgt_epi16(vb, va), va);
    mi vp = _mm256_madd_epi16(va, vb);
    acc = _mm256_add_epi64(_mm256_unpacklo_epi32(vp, zero),
      _mm256_add_epi64(acc, _mm256_unpackhi_epi32(vp, zero)));
  }
  union {ll v[4]; mi m;} u; u.m = acc; rep(i,0,4) r += u.v[i];
  for (;i<n;++i) if (a[i] < b[i]) r += a[i]*b[i]; // <- equiv
  return r;
}
```

Techniques (A)

techniques.txt 159 lines

```
Recursion
Divide and conquer
  Finding interesting points in N log N
Algorithm analysis
  Master theorem
  Amortized time complexity
Greedy algorithm
  Scheduling
  Max contiguous subvector sum
  Invariants
  Huffman encoding
Graph theory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algorithm
  MST: Prim's algorithm
  Bellman-Ford
  Konig's theorem and vertex cover
  Min-cost max flow
  Lovasz toggle
  Matrix tree theorem
  Maximal matching, general graphs
  Hopcroft-Karp
  Hall's marriage theorem
  Graphical sequences
  Floyd-Warshall
  Euler cycles
  Flow networks
  * Augmenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cut vertices, cut-edges and biconnected components
  Edge coloring
  * Trees
  Vertex coloring
  * Bipartite graphs (=> trees)
  * 3^n (special case of set cover)
  Diameter and centroid
  K'th shortest path
  Shortest cycle
Dynamic programming
  Knapsack
  Coin change
  Longest common subsequence
  Longest increasing subsequence
  Number of paths in a dag
  Shortest path in a dag
  Dynprog over intervals
  Dynprog over subsets
  Dynprog over probabilities
  Dynprog over trees
  3^n set cover
  Divide and conquer
  Knuth optimization
  Convex hull optimizations
  RMQ (sparse table a.k.a 2^k-jumps)
  Bitonic cycle
  Log partitioning (loop over most restricted)
Combinatorics
  Computation of binomial coefficients
  Pigeon-hole principle
  Inclusion/exclusion
  Catalan number
  Pick's theorem
Number theory
  Integer parts
  Divisibility
  Euclidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's little theorem
  Euler's theorem
  Phi function
  Frobenius number
  Quadratic reciprocity
  Pollard-Rho
  Miller-Rabin
  Hensel lifting
  Vieta root jumping
Game theory
  Combinatorial games
  Game trees
  Mini-max
  Nim
  Games on graphs
  Games on graphs with loops
  Grundy numbers
  Bipartite games without repetition
  General games without repetition
  Alpha-beta pruning
Probability theory
Optimization
  Binary search
  Ternary search
  Unimodality and convex functions
  Binary search on derivative
Numerical methods
  Numeric integration
  Newton's method
  Root-finding with binary/ternary search
  Golden section search
Matrices
  Gaussian elimination
  Exponentiation by squaring
Sorting
  Radix sort
Geometry
  Coordinates and vectors
  * Cross product
  * Scalar product
  Convex hull
  Polygon cut
  Closest pair
  Coordinate-compression
  Quadtrees
  KD-trees
  All segment-segment intersection
Sweeping
  Discretization (convert to events and sweep)
  Angle sweeping
  Line sweeping
  Discrete second derivatives
Strings
  Longest common substring
  Palindrome subsequences
  Knuth-Morris-Pratt
  Tries
  Rolling polynomial hashes
  Suffix array
  Suffix tree
  Aho-Corasick
  Manacher's algorithm
  Letter position lists
Combinatorial search
  Meet in the middle
  Brute-force with pruning
  Best-first (A*)
  Bidirectional search
  Iterative deepening DFS / A*
Data structures
  LCA (2^k-jumps in trees in general)
  Pull/push-technique on trees
  Heavy-light decomposition
  Centroid decomposition
  Lazy propagation
  Self-balancing trees
  Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
  Monotone queues / monotone stacks / sliding queues
  Sliding queue using 2 stacks
  Persistent segment tree
```