Contents

1. Misc

1.1. Contest

1.1.1. Makefile

```
.PRECIOUS: ./p%

%: p%
  ulimit -s unlimited && ./$<
p%: p%.cpp
  g++ -o $@ $< -std=c++17 -Wall -Wextra -Wshadow \
    -fsanitize=address,undefined
```

1.2. How Did We Get Here?

1.2.1. Macros

Use vectorizations and math optimizations at your own peril.
For gcc≥9, there are `[[likely]]` and `[[unlikely]]` attributes.
Call gcc with `-fopt-info-optimized-missed-optall` for optimization info.

```
#define _GLIBCXX_DEBUG            1 // for debug mode
#define _GLIBCXX_SANITIZE_VECTOR 1 // for asan on vectors
#pragma GCC optimize("O3", "unroll-loops")
#pragma GCC optimize("fast-math")
#pragma GCC target("avx,avx2,abm,bmi,bmi2") // tip: `lscpu`
// before a loop
#pragma GCC unroll 16 // 0 or 1 -> no unrolling
#pragma GCC ivdep
```

1.2.2. constexpr

Some default limits in gcc (7.x - trunk):

- constexpr recursion depth: 512
- constexpr loop iteration per function: 262 144
- constexpr operation count per function: 33 554 432
- template recursion depth: 900 (gcc *might* segfault first)

1.2.3. Bump Allocator

```
// global bump allocator
char mem[256 << 20]; // 256 MB
size_t rsp = sizeof mem;
void *operator new(size_t s) {
  assert(s < rsp); // MLE
  return (void *)&mem[rsp -= s];
}
void operator delete(void *) {}

// bump allocator for STL / pbds containers
char mem[256 << 20];
size_t rsp = sizeof mem;
template <typename T> struct bump {
  typedef T value_type;
  bump() {}
  template <typename U> bump(U, ...) {}
  T *allocate(size_t n) {
    rsp -= n * sizeof(T);
    rsp &= 0 - alignof(T);
    return (T *)(mem + rsp);
  }
  void deallocate(T *, size_t n) {}
};
```

1.3. Tools

1.3.1. SplitMix64

```
using ull = unsigned long long;
inline ull splitmix64(ull x) {
  // change to `static ull x = SEED;` for DRBG
  ull z = (x += 0x9E3779B97F4A7C15);
  z = (z ^ (z >> 30)) * 0xBF58476D1CE4E5B9;
  z = (z ^ (z >> 27)) * 0x94D049BB133111EB;
  return z ^ (z >> 31);
}
```

1.3.2. x86 Stack Hack

```
constexpr size_t size = 200 << 20; // 200MiB
int main() {
  register long rsp asm("rsp");
  char *buf = new char[size];
  asm("movq %0, %%rsp\n" ::"r"(buf + size));
  // do stuff
  asm("movq %0, %%rsp\n" ::"r"(rsp));
  delete[] buf;
}
```

1.4. Algorithms

1.4.1. Bit Hacks

```
// next permutation of x as a bit sequence
ull next_bits_permutation(ull x) {
  ull c = __builtin_ctzll(x), r = x + (1ULL << c);
  return (r ^ x) >> (c + 2) | r;
}
// iterate over all (proper) subsets of bitset s
void subsets(ull s) {
  for (ull x = s; x;) { --x &= s; /* do stuff */ }
}
```

1.4.2. DP opt

Aliens

```
// min dp[i] value and its i (smallest one)
pll get_dp(int cost);
ll aliens(int k, int l, int r) {
  while (l != r) {
    int m = (l + r) / 2;
    auto [f, s] = get_dp(m);
    if (s == k) return f - m * k;
    if (s < k) r = m;
    else l = m + 1;
  }
  return get_dp(l).first - l * k;
}
```

DnC DP :
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: $O((N + (hi - lo)) \log N)$

```
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); }
};
```

Knuth's Opt :
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: $O(N^2)$

1.4.3. Mo's Algorithm on Tree

```
void MoAlgoOnTree() {
  Dfs(0, -1);
  vector<int> euler(tk);
  for (int i = 0; i < n; ++i) {
    euler[tin[i]] = i;
    euler[tout[i]] = i;
  }
  vector<int> l(q), r(q), qr(q), sp(q, -1);
  for (int i = 0; i < q; ++i) {
    if (tin[u[i]] > tin[v[i]]) swap(u[i], v[i]);
    int z = GetLCA(u[i], v[i]);
    sp[i] = z[i];
    if (z == u) l[i] = tin[u[i]], r[i] = tin[v[i]];
    else l[i] = tout[u[i]], r[i] = tin[v[i]];
    qr[i] = i;
  }
  sort(qr.begin(), qr.end(), [&](int i, int j) {
    if (l[i] / kB == l[j] / kB) return r[i] < r[j];
    return l[i] / kB < l[j] / kB;
  });
  vector<bool> used(n);
  // Add(v): add/remove v to/from the path based on used[v]
  for (int i = 0, tl = 0, tr = -1; i < q; ++i) {
    while (tl < l[qr[i]]) Add(euler[tl++]);
    while (tl > l[qr[i]]) Add(euler[--tl]);
    while (tr > r[qr[i]]) Add(euler[tr--]);
    while (tr < r[qr[i]]) Add(euler[++tr]);
    // add/remove LCA(u, v) if necessary
  }
}
```

2. Data Structures

2.1. GNU PBDS

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

// most std::map + order_of_key, find_by_order, split, join
template <typename T, typename U = null_type>
using ordered_map = tree<T, U, std::less<>, rb_tree_tag,
                         tree_order_statistics_node_update>;
// useful tags: rb_tree_tag, splay_tree_tag

template <typename T> struct myhash {
  size_t operator()(T x) const; // splitmix, bswap(x*R), ...
};
// most of std::unordered_map, but faster (needs good hash)
template <typename T, typename U = null_type>
using hash_table = gp_hash_table<T, U, myhash<T>>;

// most std::priority_queue + modify, erase, split, join
using heap = priority_queue<int, std::less<>>;
// useful tags: pairing_heap_tag, binary_heap_tag,
//              (rc_)?binomial_heap_tag, thin_heap_tag
```

```
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
}
```

2.2. Line Container

```
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line &o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};
// add: line y=kx+m, query: maximum y of given x
struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b);
  }
```

```
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, 0;
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y))
      isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

2.3. Li-Chao Tree

```
constexpr ll MAXN = 2e5, INF = 2e18;
struct Line {
  ll m, b;
  Line() : m(0), b(-INF) {}
  Line(ll _m, ll _b) : m(_m), b(_b) {}
  ll operator()(ll x) const { return m * x + b; }
};
struct Li_Chao {
  Line a[MAXN * 4];
  void insert(Line seg, int l, int r, int v = 1) {
    if (l == r) {
      if (seg(l) > a[v](l)) a[v] = seg;
      return;
    }
    int mid = (l + r) >> 1;
    if (a[v].m > seg.m) swap(a[v], seg);
    if (a[v](mid) < seg(mid)) {
      swap(a[v], seg);
      insert(seg, l, mid, v << 1);
    } else insert(seg, mid + 1, r, v << 1 | 1);
  }
  ll query(int x, int l, int r, int v = 1) {
    if (l == r) return a[v](x);
    int mid = (l + r) >> 1;
    if (x <= mid)
      return max(a[v](x), query(x, l, mid, v << 1));
    else
      return max(a[v](x), query(x, mid + 1, r, v << 1 | 1));
  }
};
```

2.4. Wavelet Matrix

```
#pragma GCC target("popcnt,bmi2")
#include <immintrin.h>

// T is unsigned. You might want to compress values first
template <typename T> struct wavelet_matrix {
  static_assert(is_unsigned_v<T>, "only unsigned T");
  struct bit_vector {
    static constexpr uint W = 64;
    uint n, cnt0;
    vector<ull> bits;
    vector<uint> sum;
    bit_vector(uint n_)
        : n(n_), bits(n / W + 1), sum(n / W + 1) {}
    void build() {
      for (uint j = 0; j != n / W; ++j)
        sum[j + 1] = sum[j] + _mm_popcnt_u64(bits[j]);
      cnt0 = rank0(n);
    }
    void set_bit(uint i) { bits[i / W] |= 1ULL << i % W; }
    bool operator[](uint i) const {
      return !!(bits[i / W] & 1ULL << i % W);
    }
    uint rank1(uint i) const {
      return sum[i / W] +
             _mm_popcnt_u64(_bzhi_u64(bits[i / W], i % W));
    }
    uint rank0(uint i) const { return i - rank1(i); }
  };
  uint n, lg;
  vector<bit_vector> b;
  wavelet_matrix(const vector<T> &a) : n(a.size()) {
    lg =
    __lg(max(*max_element(a.begin(), a.end()), T(1))) + 1;
    b.assign(lg, n);
    vector<T> cur = a, nxt(n);
    for (int h = lg; h--;) {
      for (uint i = 0; i < n; ++i)
        if (cur[i] & (T(1) << h)) b[h].set_bit(i);
      b[h].build();
      int il = 0, ir = b[h].cnt0;
      for (uint i = 0; i < n; ++i)
        nxt[(b[h][i] ? ir : il)++] = cur[i];
      swap(cur, nxt);
    }
  }
  T operator[](uint i) const {
    T res = 0;
    for (int h = lg; h--;)
      if (b[h][i])
        i += b[h].cnt0 - b[h].rank0(i), res |= T(1) << h;
      else i = b[h].rank0(i);
    return res;
  }
  // query k-th smallest (0-based) in a[l, r)
  T kth(uint l, uint r, uint k) const {
    T res = 0;
    for (int h = lg; h--;) {
      uint tl = b[h].rank0(l), tr = b[h].rank0(r);
      if (k >= tr - tl) {
        k -= tr - tl;
        l += b[h].cnt0 - tl;
        r += b[h].cnt0 - tr;
        res |= T(1) << h;
      } else l = tl, r = tr;
    }
    return res;
  }
  // count of i in [l, r) with a[i] < u
  uint count(uint l, uint r, T u) const {
    if (u >= T(1) << lg) return r - l;
    uint res = 0;
    for (int h = lg; h--;) {
      uint tl = b[h].rank0(l), tr = b[h].rank0(r);
      if (u & (T(1) << h)) {
        l += b[h].cnt0 - tl;
        r += b[h].cnt0 - tr;
        res += tr - tl;
      } else l = tl, r = tr;
    }
    return res;
  }
};
```

2.5. Link-Cut Tree

```
#define l ch[0]
#define r ch[1]
template <class M> struct LCT {
  using T = typename M::T;

  struct node;
  using ptr = node *;
  struct node {
    node(int i = -1) : id(i) {}
    static inline node nil{};
    ptr p = &nil, ch[2]{&nil, &nil};
    T val = M::id(), path = M::id();
    T heavy = M::id(), light = M::id();
    bool rev = 0;
    int id;

    T sum() { return M::op(heavy, light); }

    void pull() {
      path = M::op(M::op(l->path, val), r->path);
      heavy = M::op(M::op(l->sum(), val), r->sum());
    }
    void push() {
      if (exchange(rev, 0)) l->reverse(), r->reverse();
    }
    void reverse() {
      swap(l, r), path = M::flip(path), rev ^= 1;
    }
  };
  static inline ptr nil = &node::nil;
  bool dir(ptr t) { return t == t->p->r; }
  bool is_root(ptr t) {
    return t->p == nil || (t != t->p->l && t != t->p->r);
  }
  void attach(ptr p, bool d, ptr c) {
    if (c) c->p = p;
    p->ch[d] = c, p->pull();
  }
  void rot(ptr t) {
    bool d = dir(t);
    ptr p = t->p;
    t->p = p->p;
    if (!is_root(p)) attach(p->p, dir(p), t);
    attach(p, d, t->ch[!d]);
```

```cpp
    attach(t, !d, p);
  }
  void splay(ptr t) {
    for (t->push(); !is_root(t); rot(t)) {
      ptr p = t->p;
      if (p->p != nil) p->p->push();
      p->push(), t->push();
      if (!is_root(p)) rot(dir(t) == dir(p) ? p : t);
    }
  }
  void expose(ptr t) {
    ptr cur = t, prv = nil;
    for (; cur != nil; cur = cur->p) {
      splay(cur);
      cur->light = M::op(cur->light, cur->r->sum());
      cur->light = M::op(cur->light, M::inv(prv->sum()));
      attach(cur, 1, exchange(prv, cur));
    }
    splay(t);
  }

  vector<ptr> vert;
  LCT(int n = 0) {
    for (int i = 0; i < n; i++) vert.push_back(new node(i));
  }

  void expose(int v) { expose(vert[v]); }
  void evert(int v) { expose(v), vert[v]->reverse(); }
  void link(int v, int p) {
    evert(v), expose(p);
    assert(vert[v]->p == nil);
    attach(vert[p], 1, vert[v]);
  }
  void cut(int v) {
    expose(v);
    assert(vert[v]->l != nil);
    attach(vert[v], 0, vert[v]->l->p = nil);
  }
  T get(int v) { return vert[v]->val; }
  void set(int v, const T &x) {
    expose(v), vert[v]->val = x, vert[v]->pull();
  }
  void add(int v, const T &x) {
    expose(v), vert[v]->val = M::op(vert[v]->val, x),
               vert[v]->pull();
  }
  int lca(int u, int v) {
    if (u == v) return u;
    expose(u), expose(v);
    if (vert[u]->p == nil) return -1;
    splay(vert[u]);
    return vert[u]->p != nil ? vert[u]->p->id : u;
  }
  T path_fold(int u, int v) {
    evert(u), expose(v);
    return vert[v]->path;
  }
  T subtree_fold(int v, int p) {
    evert(p), cut(v);
    T ret = vert[v]->sum();
    link(v, p);
    return ret;
  }
};
#undef l
#undef r
```

2.6. Dynamic MST

```cpp
struct Edge {
  int l, r, u, v, w;
  bool operator<(const Edge &o) const { return w < o.w; }
};
struct DynamicMST {
  int n, time = 0;
  vector<array<int, 3>> init;
  vector<Edge> edges;
  vector<int> lab, lst;
  vector<int64_t> res;
  DSU dsu1, dsu2;

  DynamicMST(vector<array<int, 3>> es, int _n)
      : n(_n), init(es), lab(n), lst(es.size()), dsu1(n),
        dsu2(n) {}

  void update(int i, int nw) {
    time++;
    auto &[u, v, w] = init[i];
    edges.push_back({lst[i], time, u, v, w});
    lst[i] = time, w = nw;
  }
  void solve(int l, int r, vector<Edge> es, int cnt,
             int64_t weight) {
    auto tmp = stable_partition(all(es), [=](auto &e) {
      return !(e.r <= l || r <= e.l);
    });
    es.erase(tmp, es.end());
    dsu1.reset(cnt), dsu2.reset(cnt);

    for (auto &e : es)
      if (l < e.l || e.r < r) dsu1.merge(e.u, e.v);
    for (auto &e : es)
      if (e.l <= l && r <= e.r && dsu1.merge(e.u, e.v))
        weight += e.w, dsu2.merge(e.u, e.v);

    if (r - l == 1) return void(res[l] = weight);
    int id = 0;
    for (int i = 0; i < cnt; i++)
      if (i == dsu2.find(i)) lab[i] = id++;
    dsu1.reset(cnt);
    for (auto &e : es) {
      e.u = lab[dsu2.find(e.u)], e.v = lab[dsu2.find(e.v)];
      if (e.l <= l && r <= e.r && !dsu1.merge(e.u, e.v))
        e.r = -1;
    }
    int m = (l + r) / 2;
    solve(l, m, es, id, weight);
    solve(m, r, es, id, weight);
  }
  auto run() { // original mst weight at res[0]
    res.resize(++time);
    for (int i = 0; i < init.size(); i++) {
      auto &[u, v, w] = init[i];
      edges.push_back({lst[i], time, u, v, w});
    }
    sort(begin(edges), end(edges));
    solve(0, time, edges, n, 0);
    return res;
  }
};
```

3. Graph

3.1. Modeling

- Maximum/Minimum flow with lower bound / Circulation problem
  1. Construct super source S and sink T.
  2. For each edge (x, y, l, u), connect $x \to y$ with capacity $u - l$.
  3. For each vertex v, denote by $in(v)$ the difference between the sum of incoming lower bounds and the sum of outgoing lower bounds.
  4. If $in(v) > 0$, connect $S \to v$ with capacity $in(v)$, otherwise, connect $v \to T$ with capacity $-in(v)$.
     - To maximize, connect $t \to s$ with capacity ∞ (skip this in circulation problem), and let f be the maximum flow from S to T. If $f \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, the maximum flow from s to t is the answer.
     - To minimize, let f be the maximum flow from S to T. Connect $t \to s$ with capacity ∞ and let the flow from S to T be f'. If $f + f' \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, f' is the answer.
  5. The solution of each edge e is $l_e + f_e$, where f_e corresponds to the flow of edge e on the graph.
- Construct minimum vertex cover from maximum matching M on bipartite graph (X, Y)
  1. Redirect every edge: $y \to x$ if $(x, y) \in M$, $x \to y$ otherwise.
  2. DFS from unmatched vertices in X.
  3. $x \in X$ is chosen iff x is unvisited.
  4. $y \in Y$ is chosen iff y is visited.
- Minimum cost cyclic flow
  1. Consruct super source S and sink T
  2. For each edge (x, y, c), connect $x \to y$ with $(cost, cap) = (c, 1)$ if $c > 0$, otherwise connect $y \to x$ with $(cost, cap) = (-c, 1)$
  3. For each edge with $c < 0$, sum these cost as K, then increase $d(y)$ by 1, decrease $d(x)$ by 1
  4. For each vertex v with $d(v) > 0$, connect $S \to v$ with $(cost, cap) = (0, d(v))$
  5. For each vertex v with $d(v) < 0$, connect $v \to T$ with $(cost, cap) = (0, -d(v))$
  6. Flow from S to T, the answer is the cost of the flow $C + K$
- Maximum density induced subgraph
  1. Binary search on answer, suppose we're checking answer T
  2. Construct a max flow model, let K be the sum of all weights
  3. Connect source $s \to v$, $v \in G$ with capacity K
  4. For each edge (u, v, w) in G, connect $u \to v$ and $v \to u$ with capacity w
  5. For $v \in G$, connect it with sink $v \to t$ with capacity $K + 2T - (\sum_{e \in E(v)} w(e)) - 2w(v)$
  6. T is a valid answer if the maximum flow $f < K|V|$
- Minimum weight edge cover
  1. For each $v \in V$ create a copy v', and connect $u' \to v'$ with weight $w(u, v)$.
  2. Connect $v \to v'$ with weight $2\mu(v)$, where $\mu(v)$ is the cost of the cheapest edge incident to v.
  3. Find the minimum weight perfect matching on G'.

- Project selection problem
  1. If $p_v > 0$, create edge (s, v) with capacity p_v; otherwise, create edge (v, t) with capacity $-p_v$.
  2. Create edge (u, v) with capacity w with w being the cost of choosing u without choosing v.
  3. The mincut is equivalent to the maximum profit of a subset of projects.
- 0/1 quadratic programming

$$\sum_x c_x x + \sum_y c_y \bar{y} + \sum_{xy} c_{xy} x\bar{y} + \sum_{xyx'y'} c_{xyx'y'}(x\bar{y} + x'\bar{y'})$$

can be minimized by the mincut of the following graph:
  1. Create edge (x, t) with capacity c_x and create edge (s, y) with capacity c_y.
  2. Create edge (x, y) with capacity c_{xy}.
  3. Create edge (x, y) and edge (x', y') with capacity $c_{xyx'y'}$.

3.2. Shortest paths

3.2.1. Dial's algorithm

```cpp
template <typename Graph>
auto dial(Graph &graph, int src, int lim) {
  vector<vector<int>> qs(lim);
  vector<int> dist(graph.size(), -1);

  dist[src] = 0;
  qs[0].push_back(src);
  for (int d = 0, maxd = 0; d <= maxd; ++d) {
    for (auto &q = qs[d % lim]; q.size();) {
      int node = q.back();
      q.pop_back();
      if (dist[node] != d) continue;
      for (auto [vec, cost] : graph[node]) {
        if (dist[vec] != -1 && dist[vec] <= d + cost)
          continue;
        dist[vec] = d + cost;
        qs[(d + cost) % lim].push_back(vec);
        maxd = max(maxd, d + cost);
      }
    }
  }
  return dist;
}
```

3.3. Matching/Flows

3.3.1. Dinic's Algorithm

```cpp
struct Dinic {
  struct edge {
    int to, cap, flow, rev;
  };
  static constexpr int MAXN = 1000, MAXF = 1e9;
  vector<edge> v[MAXN];
  int top[MAXN], deep[MAXN], side[MAXN], s, t;
  void make_edge(int s, int t, int cap) {
    v[s].push_back({t, cap, 0, (int)v[t].size()});
    v[t].push_back({s, 0, 0, (int)v[s].size() - 1});
  }
  int dfs(int a, int flow) {
    if (a == t || !flow) return flow;
    for (int &i = top[a]; i < v[a].size(); i++) {
      edge &e = v[a][i];
      if (deep[a] + 1 == deep[e.to] && e.cap - e.flow) {
        int x = dfs(e.to, min(e.cap - e.flow, flow));
        if (x) {
          e.flow += x, v[e.to][e.rev].flow -= x;
          return x;
        }
      }
    }
    deep[a] = -1;
    return 0;
  }
  bool bfs() {
    queue<int> q;
    fill_n(deep, MAXN, 0);
    q.push(s), deep[s] = 1;
    int tmp;
    while (!q.empty()) {
      tmp = q.front(), q.pop();
      for (edge e : v[tmp])
        if (!deep[e.to] && e.cap != e.flow)
          deep[e.to] = deep[tmp] + 1, q.push(e.to);
    }
    return deep[t];
  }
  int max_flow(int _s, int _t) {
    s = _s, t = _t;
    int flow = 0, tflow;
    while (bfs()) {
      fill_n(top, MAXN, 0);
      while ((tflow = dfs(s, MAXF))) flow += tflow;
    }
    return flow;
  }
  void reset() {
    fill_n(side, MAXN, 0);
    for (auto &i : v) i.clear();
  }
};
```

3.3.2. Minimum Cost Flow

```cpp
struct MCF {
  struct edge {
    ll to, from, cap, flow, cost, rev;
  } *fromE[MAXN];
  vector<edge> v[MAXN];
  ll n, s, t, flows[MAXN], dis[MAXN], pi[MAXN], flowlim;
  void make_edge(int s, int t, ll cap, ll cost) {
    if (!cap) return;
    v[s].pb(edge{t, s, cap, 0LL, cost, v[t].size()});
    v[t].pb(edge{s, t, 0LL, 0LL, -cost, v[s].size() - 1});
  }
  bitset<MAXN> vis;
  void dijkstra() {
    vis.reset();
    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vector<decltype(q)::point_iterator> its(n);
    q.push({0LL, s});
    while (!q.empty()) {
      int now = q.top().second;
      q.pop();
      if (vis[now]) continue;
      vis[now] = 1;
      ll ndis = dis[now] + pi[now];
      for (edge &e : v[now]) {
        if (e.flow == e.cap || vis[e.to]) continue;
        if (dis[e.to] > ndis + e.cost - pi[e.to]) {
          dis[e.to] = ndis + e.cost - pi[e.to];
          flows[e.to] = min(flows[now], e.cap - e.flow);
          fromE[e.to] = &e;
          if (its[e.to] == q.end())
            its[e.to] = q.push({-dis[e.to], e.to});
          else q.modify(its[e.to], {-dis[e.to], e.to});
        }
      }
    }
  }
  bool AP(ll &flow) {
    fill_n(dis, n, INF);
    fromE[s] = 0;
    dis[s] = 0;
    flows[s] = flowlim - flow;
    dijkstra();
    if (dis[t] == INF) return false;
    flow += flows[t];
    for (edge *e = fromE[t]; e; e = fromE[e->from]) {
      e->flow += flows[t];
      v[e->to][e->rev].flow -= flows[t];
    }
    for (int i = 0; i < n; i++)
      pi[i] = min(pi[i] + dis[i], INF);
    return true;
  }
  pll solve(int _s, int _t, ll _flowlim = INF) {
    s = _s, t = _t, flowlim = _flowlim;
    pll re;
    while (re.F != flowlim && AP(re.F));
    for (int i = 0; i < n; i++)
      for (edge &e : v[i])
        if (e.flow != 0) re.S += e.flow * e.cost;
    re.S /= 2;
    return re;
  }
  void init(int _n) {
    n = _n;
    fill_n(pi, n, 0);
    for (int i = 0; i < n; i++) v[i].clear();
  }
  void setpi(int s) {
    fill_n(pi, n, INF);
    pi[s] = 0;
    for (ll it = 0, flag = 1, tdis; flag && it < n; it++) {
      flag = 0;
      for (int i = 0; i < n; i++)
        if (pi[i] != INF)
          for (edge &e : v[i])
            if (e.cap && (tdis = pi[i] + e.cost) < pi[e.to])
              pi[e.to] = tdis, flag = 1;
    }
  }
```

```
};
```

3.3.3. Gomory-Hu Tree

Requires: Dinic's Algorithm

```
int e[MAXN][MAXN];
int p[MAXN];
Dinic D; // original graph
void gomory_hu() {
  fill(p, p + n, 0);
  fill(e[0], e[n], INF);
  for (int s = 1; s < n; s++) {
    int t = p[s];
    Dinic F = D;
    int tmp = F.max_flow(s, t);
    for (int i = 1; i < s; i++)
      e[s][i] = e[i][s] = min(tmp, e[t][i]);
    for (int i = s + 1; i <= n; i++)
      if (p[i] == t && F.side[i]) p[i] = s;
  }
}
```

3.3.4. Global Minimum Cut

```
// weights is an adjacency matrix, undirected
pair<int, vi> getMinCut(vector<vi> &weights) {
  int N = sz(weights);
  vi used(N), cut, best_cut;
  int best_weight = -1;

  for (int phase = N - 1; phase >= 0; phase--) {
    vi w = weights[0], added = used;
    int prev, k = 0;
    rep(i, 0, phase) {
      prev = k;
      k = -1;
      rep(j, 1, N) if (!added[j] &&
                       (k == -1 || w[j] > w[k])) k = j;
      if (i == phase - 1) {
        rep(j, 0, N) weights[prev][j] += weights[k][j];
        rep(j, 0, N) weights[j][prev] = weights[prev][j];
        used[k] = true;
        cut.push_back(k);
        if (best_weight == -1 || w[k] < best_weight) {
          best_cut = cut;
          best_weight = w[k];
        }
      } else {
        rep(j, 0, N) w[j] += weights[k][j];
        added[k] = true;
      }
    }
  }
  return {best_weight, best_cut};
}
```

3.3.5. Bipartite Minimum Cover

Requires: Dinic's Algorithm

```
// maximum independent set = all vertices not covered
// x : [0, n), y : [0, m]
struct Bipartite_vertex_cover {
  Dinic D;
  int n, m, s, t, x[maxn], y[maxn];
  void make_edge(int x, int y) { D.make_edge(x, y + n, 1); }
  int matching() {
    int re = D.max_flow(s, t);
    for (int i = 0; i < n; i++)
      for (Dinic::edge &e : D.v[i])
        if (e.to != s && e.flow == 1) {
          x[i] = e.to - n, y[e.to - n] = i;
          break;
        }
    return re;
  }
  // init() and matching() before use
  void solve(vector<int> &vx, vector<int> &vy) {
    bitset<maxn * 2 + 10> vis;
    queue<int> q;
    for (int i = 0; i < n; i++)
      if (x[i] == -1) q.push(i), vis[i] = 1;
    while (!q.empty()) {
      int now = q.front();
      q.pop();
      if (now < n) {
        for (Dinic::edge &e : D.v[now])
          if (e.to != s && e.to - n != x[now] && !vis[e.to])
            vis[e.to] = 1, q.push(e.to);
      } else {
        if (!vis[y[now - n]])
          vis[y[now - n]] = 1, q.push(y[now - n]);
        }
      }
      for (int i = 0; i < n; i++)
        if (!vis[i]) vx.pb(i);
      for (int i = 0; i < m; i++)
        if (vis[i + n]) vy.pb(i);
  }
  void init(int _n, int _m) {
    n = _n, m = _m, s = n + m, t = s + 1;
    for (int i = 0; i < n; i++)
      x[i] = -1, D.make_edge(s, i, 1);
    for (int i = 0; i < m; i++)
      y[i] = -1, D.make_edge(i + n, t, 1);
  }
};
```

3.4. Strongly Connected Components

```
template <class G> auto find_scc(G &g) {
  int n = g.size();
  vector<int> val(n), z;
  vector<char> added(n);
  vector<basic_string<int>> scc;
  int time = 0;
  auto dfs = [&](auto f, int v) -> int {
    int low = val[v] = time++;
    z.push_back(v);
    for (auto u : g[v])
      if (!added[u]) low = min(low, val[u] ?: f(f, u));
    if (low == val[v]) {
      scc.emplace_back();
      int x;
      do {
        x = z.back(), z.pop_back(), added[x] = true;
        scc.back().push_back(x);
      } while (x != v);
    }
    return val[v] = low;
  };
  for (int i = 0; i < n; i++)
    if (!added[i]) dfs(dfs, i);
  reverse(begin(scc), end(scc));
  return scc;
}
template <class G> auto condense(G &g) {
  auto scc = find_scc(g);
  int n = scc.size();
  vector<int> rep(g.size());
  for (int i = 0; i < n; i++)
    for (auto v : scc[i]) rep[v] = i;
  vector<basic_string<int>> gd(n);
  for (int v = 0; v < g.size(); v++)
    for (auto u : g[v])
      if (rep[v] != rep[u]) gd[rep[v]].push_back(rep[u]);
  for (auto &v : gd) {
    sort(begin(v), end(v));
    v.erase(unique(begin(v), end(v)), end(v));
  }
  return make_tuple(move(scc), move(rep), move(gd));
}
```

3.4.1. 2-Satisfiability

```
struct TwoSAT {
  int n;
  vector<basic_string<int>> g;

  TwoSAT(int _n) : n(_n), g(2 * n) {}

  void add_if(int x, int y) { // x => y
    g[x] += y, g[neg(y)] += neg(x);
  }
  void add_or(int x, int y) { add_if(neg(x), y); }
  void add_nand(int x, int y) { add_if(x, neg(y)); }
  void set_true(int x) { add_if(x, neg(x)); }
  void set_false(int x) { add_if(neg(x), x); }

  vector<bool> run() {
    vector<bool> res(n);
    auto [scc, id, gd] = condense(g);
    for (int i = 0; i < n; i++) {
      if (id[i] == id[neg(i)]) return {};
      res[i] = id[i] > id[neg(i)];
    }
    return res;
  }

  int neg(int x) { return x < n ? x + n : x - n; }
};
```

3.5. Manhattan Distance MST

```cpp
// returns [(dist, from, to), ...]
// then do normal mst afterwards
typedef Point<int> P;
vector<array<int, 3>> manhattanMST(vector<P> ps) {
  vi id(sz(ps));
  iota(all(id), 0);
  vector<array<int, 3>> edges;
  rep(k, 0, 4) {
    sort(all(id), [&](int i, int j) {
      return (ps[i] - ps[j]).x < (ps[j] - ps[i]).y;
    });
    map<int, int> sweep;
    for (int i : id) {
      for (auto it = sweep.lower_bound(-ps[i].y);
           it != sweep.end(); sweep.erase(it++)) {
        int j = it->second;
        P d = ps[i] - ps[j];
        if (d.y > d.x) break;
        edges.push_back({d.y + d.x, i, j});
      }
      sweep[-ps[i].y] = i;
    }
    for (P &p : ps)
      if (k & 1) p.x = -p.x;
      else swap(p.x, p.y);
  }
  return edges;
}
```

3.6. Functional graph

3.6.1. Loops

```cpp
struct Loop {
  int dist, lp_v, len;
};
template <class G> auto loops(G &f) {
  int n = f.size();
  vector<int> vis(n, n), dep(n);
  vector<Loop> res(n);
  int time = 0;
  auto dfs = [&](auto self, int v) -> int {
    vis[v] = time;
    int u = f[v];
    if (vis[u] == vis[v]) {
      int len = dep[v] - dep[u] + 1;
      res[v] = {0, v, len};
      return len - 1;
    } else if (vis[u] < vis[v]) {
      res[v] = res[u], res[v].dist++;
      return 0;
    } else {
      dep[u] = dep[v] + 1;
      int c = self(self, u);
      if (c > 0) {
        res[v] = res[u], res[v].lp_v = v;
        return c - 1;
      } else {
        res[v] = res[u], res[v].dist++;
        return 0;
      }
    }
  };
  for (int i = 0; i < n; i++, time++)
    if (vis[i] == n) dfs(dfs, i);
  return res;
}
```

4. Math

4.1. Number Theory

4.1.1. Mod Struct

A list of safe primes: 26003, 27767, 28319, 28979, 29243, 29759, 30467
910927547, 919012223, 947326223, 990669467, 1007939579, 1019126699
929760389146037459, 975500632317046523, 989312547895528379

NTT prime p	$p-1$	primitive root
65537	$1 \ll 16$	3
998244353	$119 \ll 23$	3
2748779069441	$5 \ll 39$	3
1945555039024054273	$27 \ll 56$	5

```cpp
array<int, 2> extgcd(int a, int b);

template <typename T> struct M {
  static T MOD; // change to constexpr if already known
  T v;
  M(T x = 0) {
    v = (-MOD <= x && x < MOD) ? x : x % MOD;
    if (v < 0) v += MOD;
  }
  explicit operator T() const { return v; }
  bool operator==(const M &b) const { return v == b.v; }
  bool operator!=(const M &b) const { return v != b.v; }
  M operator-() { return M(-v); }
  M operator+(M b) { return M(v + b.v); }
  M operator-(M b) { return M(v - b.v); }
  M operator*(M b) { return M((__int128)v * b.v % MOD); }
  M operator/(M b) { return *this * b.inv(); }
  // change above implementation to this if MOD is not prime
  M inv() {
    auto [x, g] = extgcd(v, MOD);
    return assert(g == 1), x < 0 ? x + MOD : x;
  }
  friend M operator^(M a, ll b) {
    M ans(1);
    for (; b; b >>= 1, a *= a)
      if (b & 1) ans *= a;
    return ans;
  }
  friend M &operator+=(M &a, M b) { return a = a + b; }
  friend M &operator-=(M &a, M b) { return a = a - b; }
  friend M &operator*=(M &a, M b) { return a = a * b; }
  friend M &operator/=(M &a, M b) { return a = a / b; }
};
using Mod = M<int>;
template <> int Mod::MOD = 1'000'000'007;
int &MOD = Mod::MOD;
```

4.1.2. Miller-Rabin

Requires: Mod Struct

```cpp
// checks if Mod::MOD is prime
bool is_prime() {
  if (MOD < 2 || MOD % 2 == 0) return MOD == 2;
  Mod A[] = {2, 7, 61}; // for int values (< 2^31)
  // ll: 2, 325, 9375, 28178, 450775, 9780504, 1795265022
  int s = __builtin_ctzll(MOD - 1), i;
  for (Mod a : A) {
    Mod x = a ^ (MOD >> s);
    for (i = 0; i < s && (x + 1).v > 2; i++) x *= x;
    if (i && x != -1) return 0;
  }
  return 1;
}
```

4.1.3. Pollard's Rho

```cpp
ll f(ll x, ll mod) { return (x * x + 1) % mod; }
// n should be composite
ll pollard_rho(ll n) {
  if (!(n & 1)) return 2;
  while (1) {
    ll y = 2, x = RNG() % (n - 1) + 1, res = 1;
    for (int sz = 2; res == 1; sz *= 2) {
      for (int i = 0; i < sz && res <= 1; i++) {
        x = f(x, n);
        res = __gcd(abs(x - y), n);
      }
      y = x;
    }
    if (res != 0 && res != n) return res;
  }
}
```

4.2. Combinatorics

4.2.1. Formulas

Derangements: $!n = (n-1)(!(n-1)+!(n-2))$

4.2.2. Stirling

```cpp
template <class T> auto stirling1(int n) {
  vector dp(n + 1, vector<T>{});
  for (int i = 0; i <= n; ++i) {
    dp[i].resize(i + 1);
    dp[i][0] = 0, dp[i][i] = 1;
    for (int j = 1; j < i; ++j)
      dp[i][j] = dp[i - 1][j - 1] + (i - 1) * dp[i - 1][j];
  }
  return dp;
}
template <class T> auto stirling2(int n) {
  vector dp(n + 1, vector<T>{});
  for (int i = 0; i <= n; ++i) {
    dp[i].resize(i + 1);
    dp[i][0] = 0, dp[i][i] = 1;
    for (int j = 1; j < i; ++j)
      dp[i][j] = dp[i - 1][j - 1] + j * dp[i - 1][j];
```

```
  }
  return dp;
}
template <class T> auto bell(int n) {
  vector<T> dp(n + 1, 0);
  auto S = stirling2<T>(n);
  for (int i = 0; i <= n; ++i)
    for (int k = 0; k <= i; ++k) dp[i] += S[i][k];
  return dp;
}
```

4.2.3. Extended Lucas

```
ll crt(vector<ll> &x, vector<ll> &mod) {
  int n = x.size();
  ll M = 1;
  for (ll m : mod) M *= m;
  ll res = 0;
  for (int i = 0; i < n; i++) {
    ll out = M / mod[i];
    res += x[i] * inv(out, mod[i]) * out;
  }
  return res;
}
ll f(ll n, ll k, ll p, ll q) {
  auto fac = [](ll n, ll p, ll q) {
    ll x = 1, y = powi(p, q);
    for (int i = 2; i <= n; i++)
      if (i % p != 0) x = x * i % y;
    return x % y;
  };
  ll r = n - k, x = powi(p, q);
  ll e0 = 0, eq = 0;
  ll mul = (p == 2 && q >= 3) ? 1 : -1;
  ll cr = r, cm = k, car = 0, cnt = 0;
  while (cr || cm || car) {
    ll rr = cr % p, rm = cm % p;
    cnt++, car += rr + rm;
    if (car >= p) {
      e0++;
      if (cnt >= q) eq++;
    }
    car /= p, cr /= p, cm /= p;
  }
  mul = powi(p, e0) * powi(mul, eq);
  ll ret = (mul % x + x) % x;
  ll tmp = 1;
  for (;; tmp *= p) {
    ret = ret * fac(n / tmp % x, p, q) % x;
    ret = ret * inv(fac(n / tmp % x, p, q), x) % x;
    ret = ret * inv(fac(n / tmp % x, p, q), x) % x;
    if (tmp > n / p && tmp > k / p && tmp > r / p) break;
  }
  return (ret % x + x) % x;
}
int comb(ll n, ll k, int m) {
  int _m = m; // can use better factorization
  vector<ll> x, mod;
  for (int p = 2; p * p <= _m; p += 1 + (p & 1)) {
    if (_m % p == 0) {
      int q = 0;
      for (; _m % p == 0; _m /= p) q++;
      x.push_back(f(n, k, p, q));
      mod.push_back(powi(p, q));
    }
  }
  if (_m > 1)
    x.push_back(f(n, k, _m, 1)), mod.push_back(_m);
  return crt(x, mod) % m;
}
```

4.3. Theorems

4.3.1. Kirchhoff's Theorem

Denote L be a $n \times n$ matrix as the Laplacian matrix of graph G, where $L_{ii} = d(i)$, $L_{ij} = -c$ where c is the number of edge (i, j) in G.

- The number of undirected spanning in G is $|\det(\tilde{L}_{11})|$.
- The number of directed spanning tree rooted at r in G is $|\det(\tilde{L}_{rr})|$.

4.3.2. Tutte's Matrix

Let D be a $n \times n$ matrix, where $d_{ij} = x_{ij}$ (x_{ij} is chosen uniformly at random) if $i < j$ and $(i, j) \in E$, otherwise $d_{ij} = -d_{ji}$. $\frac{rank(D)}{2}$ is the maximum matching on G.

4.3.3. Cayley's Formula

- Given a degree $d_1, d_2, \ldots, d_n$ for each *labeled* vertices, there are
$$\frac{(n-2)!}{(d_1-1)!(d_2-1)!\cdots(d_n-1)!}$$
spanning trees.
- Let $T_{n,k}$ be the number of *labeled* forests on n vertices with k components, such that vertex $1, 2, \ldots, k$ belong to different components. Then $T_{n,k} = kn^{n-k-1}$.

4.3.4. Erdős–Gallai Theorem

A sequence of non-negative integers $d_1 \geq d_2 \geq \ldots \geq d_n$ can be represented as the degree sequence of a finite simple graph on n vertices if and only if $d_1 + d_2 + \ldots + d_n$ is even and

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

holds for all $1 \leq k \leq n$.

4.3.5. Burnside's Lemma

Let X be a set and G be a group that acts on X. For $g \in G$, denote by X^g the elements fixed by g:

$$X^g = \{x \in X \mid gx \in X\}$$

Then

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

5. Numeric

5.1. Fast Fourier Transform

```
template <typename T>
void fft_(int n, vector<T> &a, vector<T> &rt, bool inv) {
  vector<int> br(n);
  for (int i = 1; i < n; i++) {
    br[i] = (i & 1) ? br[i - 1] + n / 2 : br[i / 2] / 2;
    if (br[i] > i) swap(a[i], a[br[i]]);
  }
  for (int len = 2; len <= n; len *= 2)
    for (int i = 0; i < n; i += len)
      for (int j = 0; j < len / 2; j++) {
        int pos = n / len * (inv ? len - j : j);
        T u = a[i + j], v = a[i + j + len / 2] * rt[pos];
        a[i + j] = u + v, a[i + j + len / 2] = u - v;
      }
  if (T minv = T(1) / T(n); inv)
    for (T &x : a) x *= minv;
}
```

Requires: Mod Struct

```
void ntt(vector<Mod> &a, bool inv, Mod primitive_root) {
  int n = a.size();
  Mod root = primitive_root ^ (MOD - 1) / n;
  vector<Mod> rt(n + 1, 1);
  for (int i = 0; i < n; i++) rt[i + 1] = rt[i] * root;
  fft_(n, a, rt, inv);
}
void fft(vector<complex<double>> &a, bool inv) {
  int n = a.size();
  vector<complex<double>> rt(n + 1);
  double arg = acos(-1) * 2 / n;
  for (int i = 0; i <= n; i++)
    rt[i] = {cos(arg * i), sin(arg * i)};
  fft_(n, a, rt, inv);
}
```

5.2. Fast Walsh-Hadamard Transform

Requires: Mod Struct

```
void fwht(vector<Mod> &a, bool inv) {
  int n = a.size();
  for (int d = 1; d < n; d <<= 1)
    for (int m = 0; m < n; m++)
      if (!(m & d)) {
        inv ? a[m] -= a[m | d] : a[m] += a[m | d]; // AND
        inv ? a[m | d] -= a[m] : a[m | d] += a[m]; // OR
        Mod x = a[m], y = a[m | d];                // XOR
        a[m] = x + y, a[m | d] = x - y;            // XOR
      }
  if (Mod iv = Mod(1) / n; inv) // XOR
    for (Mod &i : a) i *= iv;   // XOR
}
```

5.3. Subset Convolution

Requires: Mod Struct

```
#pragma GCC target("popcnt")
#include <immintrin.h>

void fwht(int n, vector<vector<Mod>> &a, bool inv) {
  for (int h = 0; h < n; h++)
    for (int i = 0; i < (1 << n); i++)
      if (!(i & (1 << h)))
        for (int k = 0; k <= n; k++)
          inv ? a[i | (1 << h)][k] -= a[i][k]
              : a[i | (1 << h)][k] += a[i][k];
}
// c[k] = sum(popcnt(i & j) == sz && i | j == k) a[i] * b[j]
vector<Mod> subset_convolution(int n, int sz,
                               const vector<Mod> &a_,
                               const vector<Mod> &b_) {
  int len = n + sz + 1, N = 1 << n;
  vector<vector<Mod>> a(1 << n, vector<Mod>(len, 0)), b = a;
  for (int i = 0; i < N; i++)
    a[i][_mm_popcnt_u64(i)] = a_[i],
    b[i][_mm_popcnt_u64(i)] = b_[i];
  fwht(n, a, 0), fwht(n, b, 0);
  for (int i = 0; i < N; i++) {
    vector<Mod> tmp(len);
    for (int j = 0; j < len; j++)
      for (int k = 0; k <= j; k++)
        tmp[j] += a[i][k] * b[i][j - k];
    a[i] = tmp;
  }
  fwht(n, a, 1);
  vector<Mod> c(N);
  for (int i = 0; i < N; i++)
    c[i] = a[i][_mm_popcnt_u64(i) + sz];
  return c;
}
```

5.4. Linear Recurrences

5.4.1. Berlekamp-Massey Algorithm

```
template <typename T>
vector<T> berlekamp_massey(const vector<T> &s) {
  int n = s.size(), l = 0, m = 1;
  vector<T> r(n), p(n);
  r[0] = p[0] = 1;
  T b = 1, d = 0;
  for (int i = 0; i < n; i++, m++, d = 0) {
    for (int j = 0; j <= l; j++) d += r[j] * s[i - j];
    if ((d /= b) == 0) continue; // change if T is float
    auto t = r;
    for (int j = m; j < n; j++) r[j] -= d * p[j - m];
    if (l * 2 <= i) l = i + 1 - l, b *= d, m = 0, p = t;
  }
  return r.resize(l + 1), reverse(r.begin(), r.end()), r;
}
```

5.4.2. Linear Recurrence Calculation

```
template <typename T> struct lin_rec {
  using poly = vector<T>;
  poly mul(poly a, poly b, poly m) {
    int n = m.size();
    poly r(n);
    for (int i = n - 1; i >= 0; i--) {
      r.insert(r.begin(), 0), r.pop_back();
      T c = r[n - 1] + a[n - 1] * b[i];
      // c /= m[n - 1];  if m is not monic
      for (int j = 0; j < n; j++)
        r[j] += a[j] * b[i] - c * m[j];
    }
    return r;
  }
  poly pow(poly p, ll k, poly m) {
    poly r(m.size());
    r[0] = 1;
    for (; k; k >>= 1, p = mul(p, p, m))
      if (k & 1) r = mul(r, p, m);
    return r;
  }
  T calc(poly t, poly r, ll k) {
    int n = r.size();
    poly p(n);
    p[1] = 1;
    poly q = pow(p, k, r);
    T ans = 0;
    for (int i = 0; i < n; i++) ans += t[i] * q[i];
    return ans;
  }
};
```

5.5. Matrices

5.5.1. Determinant

Requires: Mod Struct

```
Mod det(vector<vector<Mod>> a) {
  int n = a.size();
  Mod ans = 1;
  for (int i = 0; i < n; i++) {
    int b = i;
    for (int j = i + 1; j < n; j++)
      if (a[j][i] != 0) {
        b = j;
        break;
      }
    if (i != b) swap(a[i], a[b]), ans = -ans;
    ans *= a[i][i];
    if (ans == 0) return 0;
    for (int j = i + 1; j < n; j++) {
      Mod v = a[j][i] / a[i][i];
      if (v != 0)
        for (int k = i + 1; k < n; k++)
          a[j][k] -= v * a[i][k];
    }
  }
  return ans;
}
```

```
double det(vector<vector<double>> a) {
  int n = a.size();
  double ans = 1;
  for (int i = 0; i < n; i++) {
    int b = i;
    for (int j = i + 1; j < n; j++)
      if (fabs(a[j][i]) > fabs(a[b][i])) b = j;
    if (i != b) swap(a[i], a[b]), ans = -ans;
    ans *= a[i][i];
    if (ans == 0) return 0;
    for (int j = i + 1; j < n; j++) {
      double v = a[j][i] / a[i][i];
      if (v != 0)
        for (int k = i + 1; k < n; k++)
          a[j][k] -= v * a[i][k];
    }
  }
  return ans;
}
```

5.5.2. Solve Linear Equation

```
typedef vector<double> vd;
const double eps = 1e-12;

// solves for x: A * x = b
int solveLinear(vector<vd> &A, vd &b, vd &x) {
  int n = sz(A), m = sz(x), rank = 0, br, bc;
  if (n) assert(sz(A[0]) == m);
  vi col(m);
  iota(all(col), 0);

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

5.5.3. Freivalds' algo

Checks if $A \times B = C$ in $O(kn^2)$ with failure rate $\approx 2^{-k}$

Generate random $n \times 1$ 0/1 vector $\vec{r}$ and check: $A \times (B\vec{r}) = C\vec{r}$

5.6. Polynomial Interpolation

```
// returns a, such that a[0]x^0 + a[1]x^1 + a[2]x^2 + ...
// passes through the given points
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  rep(k, 0, n - 1) rep(i, k + 1, n) y[i] =
  (y[i] - y[k]) / (x[i] - x[k]);
  double last = 0;
  temp[0] = 1;
  rep(k, 0, n) rep(i, 0, n) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}
```

6. Geometry

6.1. Point

```
template <typename T> struct P {
  T x, y;
  P(T x = 0, T y = 0) : x(x), y(y) {}
  bool operator<(const P &p) const {
    return tie(x, y) < tie(p.x, p.y);
  }
  bool operator==(const P &p) const {
    return tie(x, y) == tie(p.x, p.y);
  }
  P operator-() const { return {-x, -y}; }
  P operator+(P p) const { return {x + p.x, y + p.y}; }
  P operator-(P p) const { return {x - p.x, y - p.y}; }
  P operator*(T d) const { return {x * d, y * d}; }
  P operator/(T d) const { return {x / d, y / d}; }
  T dist2() const { return x * x + y * y; }
  double len() const { return sqrt(dist2()); }
  P unit() const { return *this / len(); }
  friend T dot(P a, P b) { return a.x * b.x + a.y * b.y; }
  friend T cross(P a, P b) { return a.x * b.y - a.y * b.x; }
  friend T cross(P a, P b, P o) {
    return cross(a - o, b - o);
  }
};
using pt = P<ll>;
```

6.1.1. Spherical Coordinates

```
struct car_p {
  double x, y, z;
};
struct sph_p {
  double r, theta, phi;
};

sph_p conv(car_p p) {
  double r = sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
  double theta = asin(p.y / r);
  double phi = atan2(p.y, p.x);
  return {r, theta, phi};
}
car_p conv(sph_p p) {
  double x = p.r * cos(p.theta) * sin(p.phi);
  double y = p.r * cos(p.theta) * cos(p.phi);
  double z = p.r * sin(p.theta);
  return {x, y, z};
}
```

6.2. Segments

```
// for non-collinear ABCD, if segments AB and CD intersect
bool intersects(pt a, pt b, pt c, pt d) {
  if (cross(b, c, a) * cross(b, d, a) > 0) return false;
  if (cross(d, a, c) * cross(d, b, c) > 0) return false;
  return true;
}
// the intersection point of lines AB and CD
pt intersect(pt a, pt b, pt c, pt d) {
  auto x = cross(b, c, a), y = cross(b, d, a);
  if (x == y) {
    // if(abs(x, y) < 1e-8) {
    // is parallel
  } else {
    return d * (x / (x - y)) - c * (y / (x - y));
  }
}
```

6.3. Convex Hull

```
// returns a convex hull in counterclockwise order
// for a non-strict one, change cross >= to >
vector<pt> convex_hull(vector<pt> p) {
  sort(ALL(p));
  if (p[0] == p.back()) return {p[0]};
  int n = p.size(), t = 0;
  vector<pt> h(n + 1);
  for (int _ = 2, s = 0; _--; s = --t, reverse(ALL(p)))
    for (pt i : p) {
      while (t > s + 1 && cross(i, h[t - 1], h[t - 2]) >= 0)
        t--;
      h[t++] = i;
    }
  return h.resize(t), h;
}
```

6.4. Angular Sort

```
auto angle_cmp = [](const pt &a, const pt &b) {
  auto btm = [](const pt &a) {
    return a.y < 0 || (a.y == 0 && a.x < 0);
  };
  return make_tuple(btm(a), a.y * b.x, abs2(a)) <
         make_tuple(btm(b), a.x * b.y, abs2(b));
};
void angular_sort(vector<pt> &p) {
  sort(p.begin(), p.end(), angle_cmp);
}
```

6.5. Convex Polygon Minkowski Sum

```
// O(n) convex polygon minkowski sum
// must be sorted and counterclockwise
vector<pt> minkowski_sum(vector<pt> p, vector<pt> q) {
  auto diff = [](vector<pt> &c) {
    auto rcmp = [](pt a, pt b) {
      return pt{a.y, a.x} < pt{b.y, b.x};
    };
    rotate(c.begin(), min_element(ALL(c), rcmp), c.end());
    c.push_back(c[0]);
    vector<pt> ret;
    for (int i = 1; i < c.size(); i++)
      ret.push_back(c[i] - c[i - 1]);
    return ret;
  };
  auto dp = diff(p), dq = diff(q);
  pt cur = p[0] + q[0];
  vector<pt> d(dp.size() + dq.size()), ret = {cur};
  // include angle_cmp from angular-sort.cpp
  merge(ALL(dp), ALL(dq), d.begin(), angle_cmp);
  // optional: make ret strictly convex (UB if degenerate)
  int now = 0;
  for (int i = 1; i < d.size(); i++) {
    if (cross(d[i], d[now]) == 0) d[now] = d[now] + d[i];
    else d[++now] = d[i];
  }
  d.resize(now + 1);
  // end optional part
  for (pt v : d) ret.push_back(cur = cur + v);
  return ret.pop_back(), ret;
}
```

6.6. Point In Polygon

```
bool on_segment(pt a, pt b, pt p) {
  return cross(a, b, p) == 0 && dot((p - a), (p - b)) <= 0;
}
// p can be any polygon, but this is O(n)
bool inside(const vector<pt> &p, pt a) {
  int cnt = 0, n = p.size();
  for (int i = 0; i < n; i++) {
    pt l = p[i], r = p[(i + 1) % n];
    // change to return 0; for strict version
    if (on_segment(l, r, a)) return 1;
    cnt ^= ((a.y < l.y) - (a.y < r.y)) * cross(l, r, a) > 0;
  }
  return cnt;
}
```

6.6.1. Convex Version

```
// no preprocessing version
// p must be a strict convex hull, counterclockwise
// if point is inside or on border
bool is_inside(const vector<pt> &c, pt p) {
  int n = c.size(), l = 1, r = n - 1;
  if (cross(c[0], c[1], p) < 0) return false;
  if (cross(c[n - 1], c[0], p) < 0) return false;
  while (l < r - 1) {
    int m = (l + r) / 2;
```

```cpp
    T a = cross(c[0], c[m], p);
    if (a > 0) l = m;
    else if (a < 0) r = m;
    else return dot(c[0] - p, c[m] - p) <= 0;
  }
  if (l == r) return dot(c[0] - p, c[l] - p) <= 0;
  else return cross(c[l], c[r], p) >= 0;
}

// with preprocessing version
vector<pt> vecs;
pt center;
// p must be a strict convex hull, counterclockwise
// BEWARE OF OVERFLOWS!!
void preprocess(vector<pt> p) {
  for (auto &v : p) v = v * 3;
  center = p[0] + p[1] + p[2];
  center.x /= 3, center.y /= 3;
  for (auto &v : p) v = v - center;
  vecs = (angular_sort(p), p);
}
bool intersect_strict(pt a, pt b, pt c, pt d) {
  if (cross(b, c, a) * cross(b, d, a) > 0) return false;
  if (cross(d, a, c) * cross(d, b, c) >= 0) return false;
  return true;
}
// if point is inside or on border
bool query(pt p) {
  p = p * 3 - center;
  auto pr = upper_bound(ALL(vecs), p, angle_cmp);
  if (pr == vecs.end()) pr = vecs.begin();
  auto pl = (pr == vecs.begin()) ? vecs.back() : *(pr - 1);
  return !intersect_strict({0, 0}, p, pl, *pr);
}
```

6.6.2. Offline Multiple Points Version

Requires: Point, GNU PBDS

```cpp
using Double = __float128;
using Point = pt<Double, Double>;

int n, m;
vector<Point> poly;
vector<Point> query;
vector<int> ans;

struct Segment {
  Point a, b;
  int id;
};
vector&lt;Segment&gt; segs;

Double Xnow;
inline Double get_y(const Segment &u, Double xnow = Xnow) {
  const Point &a = u.a;
  const Point &b = u.b;
  return (a.y * (b.x - xnow) + b.y * (xnow - a.x)) /
         (b.x - a.x);
}
bool operator<(Segment u, Segment v) {
  Double yu = get_y(u);
  Double yv = get_y(v);
  if (yu != yv) return yu < yv;
  return u.id < v.id;
}
ordered_map&lt;Segment&gt; st;

struct Event {
  int type; // +1 insert seg, -1 remove seg, 0 query
  Double x, y;
  int id;
};
bool operator<(Event a, Event b) {
  if (a.x != b.x) return a.x < b.x;
  if (a.type != b.type) return a.type < b.type;
  return a.y < b.y;
}
vector<Event> events;

void solve() {
  set<Double> xs;
  set<Point> ps;
  for (int i = 0; i < n; i++) {
    xs.insert(poly[i].x);
    ps.insert(poly[i]);
  }
  for (int i = 0; i < n; i++) {
    Segment s{poly[i], poly[(i + 1) % n], i};
    if (s.a.x > s.b.x ||
        (s.a.x == s.b.x && s.a.y > s.b.y)) {
      swap(s.a, s.b);
    }
    segs.push_back(s);

    if (s.a.x != s.b.x) {
      events.push_back({+1, s.a.x + 0.2, s.a.y, i});
      events.push_back({-1, s.b.x - 0.2, s.b.y, i});
    }
  }
  for (int i = 0; i < m; i++) {
    events.push_back({0, query[i].x, query[i].y, i});
  }
  sort(events.begin(), events.end());
  int cnt = 0;
  for (Event e : events) {
    int i = e.id;
    Xnow = e.x;
    if (e.type == 0) {
      Double x = e.x;
      Double y = e.y;
      Segment tmp = {{x - 1, y}, {x + 1, y}, -1};
      auto it = st.lower_bound(tmp);

      if (ps.count(query[i]) > 0) {
        ans[i] = 0;
      } else if (xs.count(x) > 0) {
        ans[i] = -2;
      } else if (it != st.end() &&
                 get_y(*it) == get_y(tmp)) {
        ans[i] = 0;
      } else if (it != st.begin() &&
                 get_y(*prev(it)) == get_y(tmp)) {
        ans[i] = 0;
      } else {
        int rk = st.order_of_key(tmp);
        if (rk % 2 == 1) {
          ans[i] = 1;
        } else {
          ans[i] = -1;
        }
      }
    } else if (e.type == 1) {
      st.insert(segs[i]);
      assert((int)st.size() == ++cnt);
    } else if (e.type == -1) {
      st.erase(segs[i]);
      assert((int)st.size() == --cnt);
    }
  }
}
```

6.7. Closest Pair

```cpp
vector<pll> p; // sort by x first!
bool cmpy(const pll &a, const pll &b) const {
  return a.y < b.y;
}
ll sq(ll x) { return x * x; }
// returns (minimum dist)^2 in [l, r)
ll solve(int l, int r) {
  if (r - l <= 1) return 1e18;
  int m = (l + r) / 2;
  ll mid = p[m].x, d = min(solve(l, m), solve(m, r));
  auto pb = p.begin();
  inplace_merge(pb + l, pb + m, pb + r, cmpy);
  vector<pll> s;
  for (int i = l; i < r; i++)
    if (sq(p[i].x - mid) < d) s.push_back(p[i]);
  for (int i = 0; i < s.size(); i++)
    for (int j = i + 1;
         j < s.size() && sq(s[j].y - s[i].y) < d; j++)
      d = min(d, dis(s[i], s[j]));
  return d;
}
```

7. Strings

7.1. Knuth-Morris-Pratt Algorithm

```cpp
vector<int> pi(const string &s) {
  vector<int> p(s.size());
  for (int i = 1; i < s.size(); i++) {
    int g = p[i - 1];
    while (g && s[i] != s[g]) g = p[g - 1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}
vector<int> match(const string &s, const string &pat) {
  vector<int> p = pi(pat + '\0' + s), res;
  for (int i = p.size() - s.size(); i < p.size(); i++)
    if (p[i] == pat.size())
      res.push_back(i - 2 * pat.size());
  return res;
}
```

7.2. Suffix Array

```cpp
// sa[i]: starting index of suffix at rank i
//        0-indexed, sa[0] = n (empty string)
// lcp[i]: lcp of sa[i] and sa[i - 1], lcp[0] = 0
struct SuffixArray {
  vector<int> sa, lcp;
  SuffixArray(string &s,
              int lim = 256) { // or basic_string<int>
    int n = sz(s) + 1, k = 0, a, b;
    vector<int> x(all(s) + 1), y(n), ws(max(n, lim)),
    rank(n);
    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);
      for (int i = 0; i < n; i++)
        if (sa[i] >= j) y[p++] = sa[i] - j;
      fill(all(ws), 0);
      for (int i = 0; i < n; i++) ws[x[i]]++;
      for (int i = 1; i < lim; i++) 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 (int i = 1; i < n; i++)
        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 = 1; i < n; i++) rank[sa[i]] = i;
    for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
      for (k &&k--, j = sa[rank[i] - 1];
           s[i + k] == s[j + k]; k++);
  }
};
```

7.3. Z Value

```cpp
int z[n];
void zval(string s) {
  // z[i] => longest common prefix of s and s[i:], i > 0
  int n = s.size();
  z[0] = 0;
  for (int b = 0, i = 1; i < n; i++) {
    if (z[b] + b <= i) z[i] = 0;
    else z[i] = min(z[i - b], z[b] + b - i);
    while (s[i + z[i]] == s[z[i]]) z[i]++;
    if (i + z[i] > b + z[b]) b = i;
  }
}
```

7.4. Manacher's Algorithm

```cpp
int z[n];
void manacher(string s) {
  // z[i] => longest odd palindrome centered at i is
  //         s[i - z[i] ... i + z[i]]
  // to get all palindromes (including even length),
  // insert a '#' between each s[i] and s[i + 1]
  int n = s.size();
  z[0] = 0;
  for (int b = 0, i = 1; i < n; i++) {
    if (z[b] + b >= i)
      z[i] = min(z[2 * b - i], b + z[b] - i);
    else z[i] = 0;
    while (i + z[i] + 1 < n && i - z[i] - 1 >= 0 &&
           s[i + z[i] + 1] == s[i - z[i] - 1])
      z[i]++;
    if (z[i] + i > z[b] + b) b = i;
  }
}
```

7.5. Minimum Rotation

```cpp
int min_rotation(string s) {
  int a = 0, n = s.size();
  s += s;
  for (int b = 0; b < n; b++) {
    for (int k = 0; k < n; k++) {
      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;
}
```

7.6. Palindromic Tree

```cpp
struct palindromic_tree {
  struct node {
    int next[26], fail, len;
    int cnt,
    num; // cnt: appear times, num: number of pal. suf.
    node(int l = 0) : fail(0), len(l), cnt(0), num(0) {
      for (int i = 0; i < 26; ++i) next[i] = 0;
    }
  };
  vector<node> St;
  vector<char> s;
  int last, n;
  palindromic_tree() : St(2), last(1), n(0) {
    St[0].fail = 1, St[1].len = -1, s.pb(-1);
  }
  inline void clear() {
    St.clear(), s.clear(), last = 1, n = 0;
    St.pb(0), St.pb(-1);
    St[0].fail = 1, s.pb(-1);
  }
  inline int get_fail(int x) {
    while (s[n - St[x].len - 1] != s[n]) x = St[x].fail;
    return x;
  }
  inline void add(int c) {
    s.push_back(c -= 'a'), ++n;
    int cur = get_fail(last);
    if (!St[cur].next[c]) {
      int now = SZ(St);
      St.pb(St[cur].len + 2);
      St[now].fail = St[get_fail(St[cur].fail)].next[c];
      St[cur].next[c] = now;
      St[now].num = St[St[now].fail].num + 1;
    }
    last = St[cur].next[c], ++St[last].cnt;
  }
  inline void count() { // counting cnt
    auto i = St.rbegin();
    for (; i != St.rend(); ++i) {
      St[i->fail].cnt += i->cnt;
    }
  }
  inline int size() { // The number of diff. pal.
    return SZ(St) - 2;
  }
};
```

8. Debug List

```
- Pre-submit:
  - Did you make a typo when copying a template?
  - Test more cases if unsure.
    - Write a naive solution and check small cases.
  - Submit the correct file.

- General Debugging:
  - Read the whole problem again.
  - Have a teammate read the problem.
  - Have a teammate read your code.
    - Explain you solution to them (or a rubber duck).
  - Print the code and its output / debug output.
  - Go to the toilet.

- Wrong Answer:
  - Any possible overflows?
    - > `__int128` ?
    - Try `-ftrapv` or `#pragma GCC optimize("trapv")`
  - Floating point errors?
    - > `long double` ?
    - turn off math optimizations
    - check for `==`, `>=`, `acos(1.000000001)`, etc.
  - Did you forget to sort or unique?
  - Generate large and worst "corner" cases.
  - Check your `m` / `n`, `i` / `j` and `x` / `y`.
  - Are everything initialized or reset properly?
  - Are you sure about the STL thing you are using?
    - Read cppreference (should be available).
  - Print everything and run it on pen and paper.

- Time Limit Exceeded:
  - Calculate your time complexity again.
  - Does the program actually end?
    - Check for `while(q.size())` etc.
  - Test the largest cases locally.
  - Did you do unnecessary stuff?
    - e.g. pass vectors by value
    - e.g. `memset` for every test case
  - Is your constant factor reasonable?

- Runtime Error:
```

```
  - Check memory usage.
    - Forget to clear or destroy stuff?
    - > `vector::shrink_to_fit()`
  - Stack overflow?
  - Bad pointer / array access?
    - Try `-fsanitize=address`
  - Division by zero? NaN's?
```

9. Tech

```
- 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
```