Uzhhorod National University

UzhNU Machata

Vasyl Merenych, Oleksandr Hroskopf, Serhii Loshak

2024-03-18

Contest (1)

template.cpp

101 lines

```
// #pragma comment(linker, "/stack:200000000")
// #pragma GCC optimize("Ofast")
// #pragma GCC optimize("O3,unroll-loops")
// #pragma GCC target("sse,sse2,sse3,ssse3,sse4")
// #pragma GCC target("avx2,bmi,bmi2,popcnt,lzcnt")

// #define _GLIBCXX_DEBUG
// #define _GLIBCXX_DEBUG_PEDANTIC

#include <cassert>
#include <iomanip>
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <set>
#include <functional>
#include <array>
#include <numeric>
#include <queue>
#include <deque>
#include <cmath>
#include <climits>

using namespace std;

const int MOD = 998244353;
const long double PI = 3.141592653589793;
using ll = long long;
const ll INF = 1e18;

// #define int ll

// ---------> sashko123's defines:

#define itn int     //Vasya sorry :(
#define p_b push_back
#define fi first
#define se second
#define pii std::pair<int, int>
#define oo LLONG_MAX
#define big INT_MAX
#define elif else if

int input()
{
    int x;
    cin>>x;
    return x;
}

// ----------> end of sashko123's defines (thank you Vasya <3)

template<typename T>
using graph = vector<vector<T>>;

template<typename T>
istream& operator>>(istream& in, vector<T>& a) {
    for (auto& i: a) {
        in >> i;
    }
    return in;
}

ll fast_pow(ll a, ll b, ll mod) {
    if (b == 0)
        return 1;
    if (b % 2) {
        return (1ll * a * fast_pow(a, b - 1, mod)) % mod;
    }
    ll k = fast_pow(a, b / 2, mod);
    return (1ll * k * k) % mod;

}

ll fast_pow(ll a, ll b) {
    if (b == 0)
        return 1;
    if (b % 2) {
        return (1ll * a * fast_pow(a, b - 1));
    }
    ll k = fast_pow(a, b / 2);
    return (1ll * k * k);
}

void solve() {

}

int32_t main(int32_t argc, const char * argv[]) {
    cin.tie(0);
    cout.tie(0);
    ios_base::sync_with_stdio(0);
    // insert code here...
    int tt= 1;
    // std::cin >> tt;
    while (tt--) {
        solve();
    }
    return 0;
}
```

fast-input.h

35 lines

```
double readNumber() {
    const int BSIZE = 4096;
    static char buffer[BSIZE];
    static char* bptr = buffer + BSIZE;
    auto getChar = []() {
        if (bptr == buffer + BSIZE) {
            memset(buffer, 0, BSIZE);
            cin.read(buffer, BSIZE);
            bptr = buffer;
        }
        return *bptr++;
    };
    char c = getChar();
    while (c && (c < '0' || c > '9') && c != '-')
        c = getChar();
    bool minus = false;
    if (c == '-') minus = true, c = getChar();
    double res = 0;

    while (c >= '0' && c <= '9') {
        res = res * 10 + c - '0';
        c = getChar();
    }

      if (c == '.') {
        c = getChar();
        double cur = 0.1;
        while (c >= '0' && c <= '9') {
            res = res + (c - '0') * cur;
            c = getChar();
            cur /= 10.0;
        }
    }
    return minus ? -res : res;
}
```

.bashrc

3 lines

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++20 \
  -fsanitize=undefined,address'
xmodmap -e 'clear lock' -e 'keycode 66=less greater' #caps = <>
```

.vimrc

6 lines

```
set cin aw ai is ts=4 sw=4 tm=50 nu noeb bg=dark ru cul
sy on   |   im jk <esc>   |   im kj <esc>   |   no ; :
" Select region and then type :Hash to hash your selection.
" Useful for verifying that there aren't mistypes.
ca Hash w !cpp -dD -P -fpreprocessed \| tr -d '[:space:]' \
 \| md5sum \| cut -c-6
```

hash.sh

3 lines

```
# 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
```

Algebra (2)

xor-basis.h

Description: Xor basis, all elements in the main set can be constructed using xor operation and elements in the basis

Time: insert per element - $\mathcal{O}(log(A_max))$

<vector> hash46 lines

```
template<typename T = int, int max_bit = 31>
struct xor_basis
{
  std::vector<T> basis; // basis[i] -> element with smallest
       set bit equal to i
  int sz; // Current size of the basis

  xor_basis() {
    basis.assign(max_bit);
  }

  bool insert(T val) {
    for (int i = 0; i < max_bit; i++) {
      if (((val >> i)&1) == 0)
        continue;
      if (!basis[i]) {
        basis[i] = val;
        sz++;
        return true;
      }
      val ^= basis[i];
    }
    return false;
  }

  bool contains(T val) {
    for (int i = 0; i < max_bit; i++) {
      if (((val >> i)&1) == 0)
        continue;
      if (!basis[i]) {
        return false;
      }
      val ^= basis[i];
    }
```

```
    return true;
  }

  T max_element() { // not-sure
    T val = 0;
    for (int i = max_bit - 1; i >= 0; i--) {
      if (basis[i] && !((val>>i)&1)) {
        val ^= basis[i];
      }
    }
    return val;
  }
};
```

fft.h

Description: FFT implementation
Time: (n + m) * log(n + m)

<vector>, <complex> hash56 lines

```
using cd = std::complex<double>;

void fft(std::vector<cd> & a, bool invert) {
    int n = a.size();

    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

        if (i < j)
            swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd & x : a)
            x /= n;
    }
}

template<typename T>
std::vector<T> multiply(const std::vector<T>& a, const std::
     vector<T>& b) {
    std::vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end
         ());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    std::vector<T> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    result.resize(a.size() + b.size() - 1);
    return result;
}
```

nnt.h

Description: NNT implementation by modulo 998244353
Time: (n + m) * log(n + m)

<vector> hash68 lines

```
const int root = 31; // primitive root of module
const int root_1 = fast_pow(root, MOD - 2, MOD); // (primitive
     root) ^ -1
const int root_pw = 1 << 23; // max power of 2 in MOD - 1

inline int inverse(int n, int mod) {
  return fast_pow(n, mod - 2, mod);
}

void nnt(std::vector<int> &a, bool invert, int mod) {
    int n = a.size();

    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

        if (i < j)
            std::swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        int wlen = invert ? root_1 : root;
        for (int i = len; i < root_pw; i <<= 1)
            wlen = (int)(1LL * wlen * wlen % mod);

        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i+j], v = (int)(1LL * a[i+j+len/2] *
                     w % mod);
                a[i+j] = u + v < mod ? u + v : u + v - mod;
                a[i+j+len/2] = u - v >= 0 ? u - v : u - v + mod
                     ;
                w = (1ll * w * wlen % mod);
            }
        }
    }

    if (invert) {
        int n_1 = inverse(n, mod);
        for (int& x : a)
            x = (1ll * x * n_1 % mod);
    }
}

std::vector<int> multiply(const std::vector<int>& a, const std
     ::vector<int>& b, int mod) {
  int n = a.size(), m = b.size();
  int k = n + m - 1;
  int l = 1;
  while (l < k)
    l <<= 1;

  std::vector<int> A(l), B(l);
  for (int i = 0; i < n; i++) {
    A[i] = a[i];
  }
  for (int i = 0; i < m; i++) {
    B[i] = b[i];
  }

  nnt(A, false, mod);
  nnt(B, false, mod);
  for (int i = 0; i < B.size(); i++) {
    A[i] = (1ll * A[i] * B[i]) % mod;
  }
  nnt(A, true, mod);
  A.resize(k);
  return A;
}
```

floor-sum.h

Description:

$$sum_{i=0}^{n-1}$$

floor((a * i + b) / m)
Time: log(n)

<utility> hash24 lines

```
using ull = unsigned long long;

ull floor_sum_unsigned(ull n, ull m, ull a, ull b) {
    ull ans = 0;
    while (true) {
        if (a >= m) {
            ans += n * (n - 1) / 2 * (a / m);
            a %= m;
        }
        if (b >= m) {
            ans += n * (b / m);
            b %= m;
        }

        ull y_max = a * n + b;
        if (y_max < m) break;
        // y_max < m * (n + 1)
        // floor(y_max / m) <= n
        n = (ull)(y_max / m);
        b = (ull)(y_max % m);
        std::swap(m, a);
    }
    return ans;
}
```

berlekamp-massey.h

Description: For given n first elements of sequence a, return array c, a[i] = sum(j = 1 ... —c—) a[i-j] * c[j])
Time: $\mathcal{O}(n^2)$

hash45 lines

```
template<typename T>
vector<T> berlekamp_massey(const vector<T> &s) {
    vector<T> c;
    vector<T> oldC;
    int f = -1;
    for (int i=0; i<(int)s.size(); i++) {
        T delta = s[i];
        for (int j=1; j<=(int)c.size(); j++)
            delta -= c[j-1] * s[i-j];
        if (delta == 0)
            continue;
        if (f == -1) {

            c.resize(i + 1);
            mt19937 rng(chrono::steady_clock::now().
                 time_since_epoch().count());
```

```
            for (T &x : c)
                x = rng();
            f = i;
        } else {
            vector<T> d = oldC;
            for (T &x : d)
                x = -x;
            d.insert(d.begin(), 1);
            T df1 = 0;
            for (int j=1; j<=(int)d.size(); j++)
                df1 += d[j-1] * s[f+1-j];
            assert(df1 != 0);
            T coef = delta / df1;
            for (T &x : d)
                x *= coef;
            vector<T> zeros(i - f - 1);
            zeros.insert(zeros.end(), d.begin(), d.end());
            d = zeros;
            vector<T> temp = c;
            c.resize(max(c.size(), d.size()));
            for (int j=0; j<(int)d.size(); j++)
                c[j] += d[j];
            if (i - (int) temp.size() > f - (int) oldC.size())
                {
                oldC = temp;
                f = i;
            }
        }
    }
    return c;
}
```

Numeric (3)

primitive-root.h

Description: Primitive root of n

hash54 lines

```
constexpr long long safe_mod(long long x, long long m) {
    x %= m;
    if (x < 0) x += m;
    return x;
}

long long pow_mod(long long x, long long n, int m) {
    if (m == 1) return 0;
    unsigned int _m = (unsigned int)(m);
    unsigned long long r = 1;
    unsigned long long y = safe_mod(x, m);
    while (n) {
        if (n & 1) r = (r * y) % _m;
        y = (y * y) % _m;
        n >>= 1;
    }
    return r;
}

int primitive_root(int m) {
    if (m == 2) return 1;
    if (m == 167772161) return 3;
    if (m == 469762049) return 3;
    if (m == 754974721) return 11;
    if (m == 998244353) return 3;
    int divs[20] = {};
    divs[0] = 2;
    int cnt = 1;
    int x = (m - 1) / 2;
    while (x % 2 == 0) x /= 2;
    for (int i = 3; (long long)(i)*i <= x; i += 2) {
        if (x % i == 0) {
            divs[cnt++] = i;
            while (x % i == 0) {
                x /= i;
            }
        }
    }
    if (x > 1) {
        divs[cnt++] = x;
    }
    for (int g = 2;; g++) {
        bool ok = true;
        for (int i = 0; i < cnt; i++) {
            if (pow_mod(g, (m - 1) / divs[i], m) == 1) {
                ok = false;
                break;
            }
        }
        if (ok) return g;
    }
}

template <int m> constexpr int primitive_root = primitive_root(
    m);
```

pollard-rho.h

Description: Finds divider of N.
Time: $\mathcal{O}\left(N^(1/4) * log(N)\right)$

<numeric> hash20 lines

```
long long mult(long long a, long long b, long long mod) {
    return (__int128)a * b % mod;
}

long long f(long long x, long long c, long long mod) {
    return (mult(x, x, mod) + c) % mod;
}

long long rho(long long n, long long x0=2, long long c=1) {
    long long x = x0;
    long long y = x0;
    long long g = 1;
    while (g == 1) {
        x = f(x, c, n);
        y = f(y, c, n);
        y = f(y, c, n);
        g = std::gcd(abs(x - y), n);
    }
    return g;
}
```

miller-rabin.h

Description: checks whether given number (up to 1e18) is prime
Time: +- $\mathcal{O}\left(log^3(n)\right)$

hash43 lines

```
using u128 = __uint128_t;

ll fast_pow(ll a, ll b, ll mod) {
    if (b == 0)
        return 1;
    if (b % 2) {
        return ((u128) a * fast_pow(a, b - 1, mod)) % mod;
    }
    ll k = fast_pow(a, b / 2, mod);
    return ((u128) k * k) % mod;
}

bool check_composite(ll n, ll a, ll d, int s) {
    ll x = fast_pow(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = (u128)x * x % n;
        if (x == n - 1)
            return false;
    }
    return true;
};

bool miller_rabin(ll n) {
    if (n < 2)
        return false;

    int r = 0;
    ll d = n - 1;
    while ((d & 1) == 0) {
        d >>= 1;
        r++;
    }

    for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37})
        {
        if (n == a)
            return true;
        if (check_composite(n, a, d, r))
            return false;
    }
    return true;
}
```

euclidean-algorithm.h

Description: Find x and y, s.t. x*a + y*b = gcd(a, b) // extended Euclidean algorithm
Time: $\mathcal{O}\left(logmin(a,b)\right)$

<tuple> hash12 lines

```
template<typename T>
std::pair<T, T> euclidean_algorithm(T a, T b) {
    T x = 1, y = 0;
    T x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        T q = a1 / b1;
        std::tie(x, x1) = std::make_tuple(x1, x - q * x1);
        std::tie(y, y1) = std::make_tuple(y1, y - q * y1);
        std::tie(a1, b1) = std::make_tuple(b1, a1 - q * b1);
    }
    return {x, y};
}
```

chinese-remainder-theorem.h

Description: Solves chinese remainder theorem.
Time: $\mathcal{O}(n)$, n - number of equations.

hash19 lines

```
struct Congruence {
    long long a, m;
};

long long chinese_remainder_theorem(vector<Congruence> const&
     congruences) {
    long long M = 1;
    for (auto const& congruence : congruences) {
        M *= congruence.m;
    }

    long long solution = 0;
    for (auto const& congruence : congruences) {
        long long a_i = congruence.a;
        long long M_i = M / congruence.m;
        long long N_i = mod_inv(M_i, congruence.m);
```

```
        solution = (solution + a_i * M_i % M * N_i) % M;
    }
    return solution;
}
```

Data structures (4)

fenwick-tree.h

Description: Fenwick Tree, update(+=) at element, sum at segment. R is excluded.
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

<cassert>, <vector> hash32 lines

```
template <class T> struct fenwick_tree {
  public:
    fenwick_tree() : _n(0) {}
    fenwick_tree(int n) : _n(n), data(n) {}

    void add(int p, T x) {
        assert(0 <= p && p < _n);
        p++;
        while (p <= _n) {
            data[p - 1] += T(x);
            p += p & -p;
        }
    }

    T sum(int l, int r) {
        assert(0 <= l && l <= r && r <= _n);
        return sum(r) - sum(l);
    }

  private:
    int _n;
    std::vector<T> data;

    T sum(int r) {
        T s = 0;
        while (r > 0) {
            s += data[r - 1];
            r -= r & -r;
        }
        return s;
    }
};
```

fenwick-tree-2d.h

Description: 2d fenwick tree, update(+=) at element, sum at 2d segment. R is excluded.
Time: update - $\mathcal{O}(\log N * \log M)$, get - $\mathcal{O}(\log N * \log M)$ hash53 lines

```
template <class T> struct fenwick_tree_2d{
    struct fenwick_tree {
        int n;
        unordered_map<int, T> data;
        fenwick_tree(): n(0) {};
        fenwick_tree(int n): n(n) {};

        void add(int p, T x) {
            assert(0 <= p && p < n);
            p++;
            while (p <= n) {
                data[p - 1] += T(x);
                p += p & -p;
            }
        }

        T pref_sum(int r){
            T s = 0;
            while (r > 0) {
                s += data[r - 1];
                r -= r & -r;
            }
            return s;
        }
    };

  int n, m;
  std::vector<fenwick_tree> data;

  fenwick_tree_2d(int n,int m): n(n), m(m), data(n,
      fenwick_tree(m)) {};

  void add(int x, int y, T val) {
        assert(0 <= x && x < n);
    x++;
        while (x <= n) {
            data[x - 1].add(y, val);
            x += x & -x;
        }
  }

  T pref_sum(int xr, int yr){
        T s = 0;
        while (xr > 0) {
            s += data[xr - 1].pref_sum(yr);
            xr -= xr & -xr;
        }
        return s;
    }

    T sum(int xl, int yl, int xr, int yr) {
        return pref_sum(xr, yr) - pref_sum(xr, yl) - pref_sum(
            xl, yr) + pref_sum(xl, yl);
    }
};
```

segment-tree.h

Description: Segment tree, update(+=) at element, sum at segment. R is excluded.
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

<array> hash38 lines

```
template<typename T, int N = (1 << 18)> struct segment_tree {
  std::array<T, 2 * N> tree;
  segment_tree() {
    tree.fill(T());
  }

  void update(int pos, T val) {
    pos += N;
    tree[pos] += val;
    pos >>= 1;
    while (pos > 0) {
      tree[pos] = tree[pos << 1] + tree[(pos << 1) | 1];
      pos >>= 1;
    }
  }

  T get_sum(int l, int r) {
    l += N;
    r += N;

    T ans = T();
    while (l < r) {
      if (l & 1) {
        ans += tree[l++];
      }
      if (r & 1) {
        ans += tree[--r];
      }
      l >>= 1;
      r >>= 1;
    }
    return ans;
  }

  T get(int pos) {
    return tree[N + pos];
  }
};
```

lazy-segment-tree.h

Description: Segment tree, update(+=) at segment, sum at segment. R is excluded.
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

<array>, <vector> hash82 lines

```
template<typename T> struct lazy_segment_tree {
  struct node{
    T sum = 0;
    T promise = 0;
  };

  int n;
  std::vector<node> tree;

  lazy_segment_tree(int n_, const vector<T>& init): n(n_) {
    tree.assign(4 * n, node{});
    build(1, 0, n, init);
  }

  void build(int v, int l, int r, const vector<T>& init) {
    if (l + 1 == r) {
      tree[v].sum = init[l];
      return;
    }
    if (l >= r)
      return;
    int mid = (l + r) / 2;
    build(2 * v, l, mid, init);
    build(2 * v + 1, mid, r, init);

    tree[v].sum = (tree[2 * v].sum + tree[2 * v + 1].sum);
  }

  void update(int l, int r, T value) {
    update(1, 0, n, l, r, value);
  }

  T get(int l, int r) {
    return get(1, 0, n, l, r);
  }

  T get(int pos) {
    return get(1, 0, n, pos, pos + 1);
  }

  void push(int v, int l, int r) {
    if (tree[v].promise == 0)
      return;

    tree[v].sum += tree[v].promise * (r - l);
    if (l + 1 < r) {
      tree[2 * v].promise += tree[v].promise;
      tree[2 * v + 1].promise += tree[v].promise;
    }
    tree[v].promise = 0;
  }
```

```
  void update(int v, int tl, int tr, int l, int r, T value) {
    push(v, tl, tr);
    if (l >= tr || tl >= r)
      return;
    if (l <= tl && tr <= r) {
      tree[v].promise += value;
      push(v, tl, tr);
      return;
    }
    int mid = (tl + tr) / 2;
    update(2 * v, tl, mid, l, r, value);
    update(2 * v + 1, mid, tr, l, r, value);

    tree[v].sum = tree[2 * v].sum + tree[2 * v + 1].sum;
  }

  T get(int v, int tl, int tr, int l, int r) {
    push(v, tl, tr);
    if (l >= tr || tl >= r)
      return 0;
    if (l <= tl && tr <= r) {
      return tree[v].sum;
    }
    int mid = (tl + tr) / 2;
    auto left = get(2 * v, tl, mid, l, r);
    auto right = get(2 * v + 1, mid, tr, l, r);

    return left + right;
  }
};
```

persistent-segment-tree.h

Description: Persistent segment tree, update(+=) at segment, sum at segment. R is excluded.
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

<array> hash83 lines

```
template<typename T> struct persistent_segment_tree {
  struct node {
    T sum;
    int left, right;

    node(T val = T()): sum(val), left(-1), right(-1) {}
    node(int left_, int right_): sum(T()), left(left_), right(
        right_) {}
  };
  std::vector<node> tree;

  node create_node(T val = 0) {
    return node(val);
  }
  node create_node(int left, int right) {
    auto v = node(left, right);
    v.sum = (left != -1 ? tree[left].sum : 0) + (right != -1 ?
        tree[right].sum : 0);

    return v;
  }
  int LAST = 0;
  template<typename... params>
  int new_node(params... args) {
    tree[++LAST] = create_node(args...);
    return LAST;
  }
  int n;
  persistent_segment_tree(int n_, int sz_, const vector<T>& a,
      int& first_root): n(n_) {
    tree.resize(sz_);
    first_root = build(0, n, a);
  }

  int build(int l, int r, const vector<T>& a) {
    if (l + 1 == r) {
      return new_node(a[l]);
    }
    int mid = (r + l) / 2;
    int left = build(l, mid, a);
    int right = build(mid, r, a);
    return new_node(left, right);
  }

  int update(int root, int pos, T val) {
    return update(root, 0, n, pos, val);
  }

  T get(int root, int l, int r) {
    return get(root, 0, n, l, r);
  }

  T get(int root, int pos) {
    return get(root, 0, n, pos, pos + 1);
  }

  int update(int v, int l, int r, int pos, T val) {
    if (l + 1 == r) {
      return new_node(val + tree[v].sum);
    }

    int mid = (l + r) / 2;
    if (pos < mid) {
      return new_node(
        update(tree[v].left, l, mid, pos, val),
        tree[v].right
      );
    } else {
      return new_node(
        tree[v].left,
        update(tree[v].right, mid, r, pos, val)
      );
    }
  }

  T get(int v, int tl, int tr, int l, int r) {
    if (tr <= l || r <= tl)
      return 0;
    if (l <= tl && tr <= r) {
      return tree[v].sum;
    }

    int mid = (tl + tr) / 2;
    return get(tree[v].left, tl, mid, l, r) + get(tree[v].right
        , mid, tr, l, r);
  }
};
```

li-chao-tree.h

Description: Li-Chao tree, online convex hull for maximizing f(x) = k * x + b, for minimization use (-k) * x + (-b)
Time: add - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

hash74 lines

```
template<typename T> struct li_chao_tree {
  const T MX = 1e9 + 1;
  struct line {
    T k = 0;
    T b = -INF;

    T f(T x) const {
      return k * x + b;
    }
  };
  struct node {
    line ln;
    node* left = nullptr;
    node* right = nullptr;
  };

  node* new_node() {
    const int N = 100000;
    static node* block;
    static int count = N;

    if (count == N) {
      block= new node[N];
      count = 0;
    }
    return (block + count++);
  };

  node* root = new_node();

  T get(T x) {
    return get(root, -MX, MX, x);
  }
    void add(line ln) {
        add(root, -MX, MX, ln);
    }

  T get(node*& v, T l, T r, T x) {
    if (!v || l > r) {
      return -INF;
    }
    T ans = v->ln.f(x);
    if (r == l) {
      return ans;
    }
    T mid = (r + l) / 2;
    if (x <= mid) {
      return max(ans, get(v->left, l, mid, x));
    } else {
            return max(ans, get(v->right, mid + 1, r, x));
        }
  }

    void add(node*& v, T l, T r, line ln) {
    if (l > r)
      return;
        if (!v) {
            v = new_node();
        }
    T m = (r + l) / 2;
        bool left = v->ln.f(l) < ln.f(l);
    bool md = v->ln.f(m) < ln.f(m);
        if (md)
            swap(v->ln, ln);
        if (l == r) {
            return;
        }
        if (left != md) {
            add(v->left, l, m, ln);
        } else {
            add(v->right, m + 1, r, ln);
        }
    }
};
```

rope.h

Description: Rope data structure
Time: all get queries $\mathcal{O}(\log N)$

<ext/rope> hash11 lines

```
using namespace __gnu_cxx;

rope<int> v;
for(int i = 1; i <= n; ++i)
  v.push_back(i);
rope<int> cur = v.substr(l, r - l + 1); // start, length
v.erase(l, r - l + 1); // start, length
v.insert(v.mutable_begin(), cur);

for(rope <int>::iterator it = v.mutable_begin(); it != v.
    mutable_end(); ++it)
  cout << *it << " ";
```

ordered-set.h

Description: A red-black tree with the ability to get an element by index (find_by_order) and index of a specific element (order_of_key)
Time: get - $\mathcal{O}(\log N)$, segment tree is 2 times faster

<ext/pb_ds/assoc_container.hpp>, <ext/pb_ds/tree_policy.hpp> hash4 lines

```
using namespace __gnu_pbds;

template<typename T>
using ordered_set = tree<T,null_type,less<T>,rb_tree_tag,
    tree_order_statistics_node_update>;
```

sparse-table.h

Description: Min sparse table.
Time: build - $\mathcal{O}(N * \log N)$, get - $\mathcal{O}(1)$

<cassert>, <array>, <vector> hash29 lines

```
template<typename T>
struct sparse_table {
    static const int K = 20;
    std::array<std::vector<T>, K> ar;
    std::vector<int> lg;
    int n;

    sparse_table(int n, const vector<T>& a): n(n) {
        lg.resize(n + 1);
        lg[1] = 0;
        for (int i = 2; i <= n; i++)
            lg[i] = lg[i >> 1] + 1;

        ar[0] = a;

        for (int k = 0; k + 1 < K; k++) {
            ar[k + 1].resize(n);
            for (int i = 0; i + (1 << k) < n; i++) {
                ar[k + 1][i] = min(ar[k][i], ar[k][i + (1 << k)
                    ]);
            }
        }
    }

    T get(int l, int r) {
        assert(0 <= l < n && l < r && r <= n);
        int power = lg[r - l];
        return min(ar[power][l], ar[power][r - (1 << power)]);
    }
};
```

xor-trie.h

Description: Binary t for integer numbers. Get finds maximum xor of two numbers.
Time: add - $\mathcal{O}(\log A)$, get - $\mathcal{O}(\log A)$

<cassert>, <array> hash35 lines

```
struct xor_trie_node {
  int cnt = 0;
  std::array<xor_trie_node*, 2> mp = {nullptr, nullptr};

  void add(int mask, int k = 30) {
    cnt++;
    if (k == -1)
      return;
    int bit = (mask>>k)&1;

    if (!mp[bit])
      mp[bit] = new xor_trie_node();
    mp[bit]->add(mask, k - 1);
  }

  void remove(int mask, int k = 30) {
    cnt--;
    if (k == -1)
      return;
    int bit = (mask>>k)&1;

    assert(mp[bit] && mp[bit]->cnt > 0);
    mp[bit]->remove(mask, k - 1);
  }

  int get(int mask, int k = 30) {
    if (k == -1)
      return 0;
    int bit = (mask>>k)&1;
    int cur= bit;
    if (mp[!bit] && mp[!bit]->cnt)
      cur = !bit;
    return ((cur^bit) << k) | mp[cur]->get(mask, k - 1);
  }
};
```

Graphs (5)

articulation-points.h

Description: Finds all articulation points.
Time: $\mathcal{O}(N + M)$

<set>, <vector>, <functional> hash35 lines

```
template<typename T>
using graph = std::vector<std::vector<T>>;

std::set<int> find_articulation_points(int n, graph<int> g) {
    std::vector<int> used(n), tin(n), fup(n);
    int T = 0;
    std::set<int> nodes;
    std::function<void(int, int)> dfs = [&](int v, int p = -1)
        {
        used[v] = true;
        tin[v] = fup[v] = T++;
        int cnt = 0;
        for (auto to : g[v]) {
            if (to == p)
                continue;
            if (used[to]) {
                fup[v] = std::min(fup[v], tin[to]);
            } else {
                dfs(to, v);
                fup[v] = std::min(fup[v], fup[to]);
                if (fup[to] >= tin[v] && p != -1) {
                    nodes.insert(v);
                }
                cnt++;
            }
        }

        if (cnt > 1 && p == -1)
            nodes.insert(v);
    };
    for (int i = 0; i < n; i++) {
        if (!used[i])
            dfs(i, -1);
    }
    return nodes;
}
```

bridges.h

Description: Finds all bridges in the undirected graph.
Time: $\mathcal{O}(N + M)$

<vector>, <functional> hash30 lines

```
template<typename T>
using graph = std::vector<std::vector<T>>;

std::vector<std::pair<int, int>> find_bridges(int n, graph<int>
     g) {
    std::vector<int> used(n), tin(n), fup(n);
    int T = 0;
    std::vector<std::pair<int, int>> edges;
    std::function<void(int, int)> dfs = [&](int v, int p = -1)
        {
        used[v] = true;
        tin[v] = fup[v] = T++;
        for (auto to : g[v]) {
            if (to == p)
                continue;
            if (used[to]) {
                fup[v] = std::min(fup[v], tin[to]);
            } else {
                dfs(to, v);
                fup[v] = std::min(fup[v], fup[to]);
                if (fup[to] ==  tin[to]) {
                    edges.push_back({v, to});
                }
            }
        }
    };
    for (int i = 0; i < n; i++) {
        if (!used[i])
            dfs(i, -1);
    }
    return edges;
}
```

dsu.h

Description: Disjoint Set Union
Time: $\mathcal{O}(n)$ time (amortized) by sizes

<algorithm>, <cassert>, <vector> hash59 lines

```
struct dsu {
  public:
    dsu() : _n(0) {}
    dsu(int n) : _n(n), parent_or_size(n, -1) {}

    int merge(int a, int b) {
        assert(0 <= a && a < _n);
        assert(0 <= b && b < _n);
        int x = leader(a), y = leader(b);
        if (x == y) return x;
        if (-parent_or_size[x] < -parent_or_size[y]) std::swap(
            x, y);
        parent_or_size[x] += parent_or_size[y];
```

```
        parent_or_size[y] = x;
        return x;
    }

    bool same(int a, int b) {
        assert(0 <= a && a < _n);
        assert(0 <= b && b < _n);
        return leader(a) == leader(b);
    }

    int leader(int a) {
        assert(0 <= a && a < _n);
        if (parent_or_size[a] < 0) return a;
        return parent_or_size[a] = leader(parent_or_size[a]);
    }

    int size(int a) {
        assert(0 <= a && a < _n);
        return -parent_or_size[leader(a)];
    }

    std::vector<std::vector<int>> groups() {
        std::vector<int> leader_buf(_n), group_size(_n);
        for (int i = 0; i < _n; i++) {
            leader_buf[i] = leader(i);
            group_size[leader_buf[i]]++;
        }
        std::vector<std::vector<int>> result(_n);
        for (int i = 0; i < _n; i++) {
            result[i].reserve(group_size[i]);
        }
        for (int i = 0; i < _n; i++) {
            result[leader_buf[i]].push_back(i);
        }
        result.erase(
            std::remove_if(result.begin(), result.end(),
                           [&](const std::vector<int>& v) {
                               return v.empty(); }),
            result.end());
        return result;
    }

  private:
    int _n;
    // root node: -1 * component size
    // otherwise: parent
    std::vector<int> parent_or_size;
};
```

dynamic-connectivity-problem.h

Description: Disjoint Set Union with roolbacks
Time: $\mathcal{O}(q * log(q) * log(n))$, q - number of queries, n - number of vertices

<array>, <vector> hash110 lines

```
struct Query {
    int v, u;
    bool united;
    Query(int _v, int _u) : v(_v), u(_u) {}
};

struct DSU
{
  struct DSU_save
  {
    int v, rang_v;
    int u, rang_u;
  };
  int n;
  std::vector<int> pred, rang;
  std::vector<DSU_save> saves;

  DSU(int n_): n(n_) {
    pred.resize(n);
    rang.resize(n);
    for (int i = 0; i < n; i++) {
      pred[i] = i;
      rang[i] = 0;
    }
  }
  int get(int v) {
    if (pred[v] == v) {
      return v;
    }
    return get(pred[v]);
  }

  bool merge(int u, int v) {
    u = get(u);
    v = get(v);
    if (u == v)
      return false;
    if (rang[u] < rang[v])
      std::swap(u, v);
    saves.push_back({v, rang[v], u, rang[u]});
    pred[v] = u;

    if (rang[u] == rang[v]) {
      rang[u]++;
    }
    return true;
  }

  void rollback() {
    if (saves.empty())
      return;
    auto [v, rang_v, u, rang_u] = saves.back();
    rang[v] = rang_v;
    rang[u] = rang_u;

    pred[v] = v;
    pred[u] = u;

    saves.pop_back();
  }
};

struct dynamic_connectivity_problem {
  std::vector<std::vector<Query>> tree;
  int q, n;
  DSU dsu;
  dynamic_connectivity_problem(int q_, int n_): q(q_), n(n_),
       dsu(DSU(n_)) {
    tree.resize(4 * q);
  }

  void add(Query a, int l, int r) {
    add(1, 0, q, l, r, a);
  }

  void add(int v, int tl, int tr, int l, int r, const Query& a)
       {
    if (l <= tl && tr <= r) {
      tree[v].push_back(a);
      return;
    }
    if (l >= tr || tl >= r)
      return;
    int mid = (tr + tl) / 2;
    add(2 * v, tl, mid, l, r, a);
    add(2 * v + 1, mid, tr, l, r, a);
  }

  void dfs(int v, int l, int r) {
    if (l >= r)
      return;

    for (auto& q: tree[v]) {
      q.united = dsu.merge(q.u, q.v);
    }
    if (l + 1 == r) {
      int x = dsu.get(0);
      // do something
    } else {
      int mid = (r + l) / 2;
      dfs(2 * v, l, mid);
      dfs(2 * v + 1, mid, r);
    }

    for (auto& q: tree[v]) {
      if (q.united)
        dsu.rollback();
      q.united = false;
    }
  }
};
```

kuhn-matching.h

Description: Fast pair matching algorithm. To find the minimum vertex cover start dfs from each vertice in the left that is not in maximum matching, from the left side choce unvisited vertices and from right chose visited.
Time: $\mathcal{O}(n * (n + m))$

<vector>, <utility> hash44 lines

```
using namespace std;

template<typename T>
using graph = vector<vector<T>>;

bool dfs(int v, graph<int>& g, vector<int>& mt, vector<int>&
     rev_mt, vector<int>& used) {
    if (used[v] == 1)
        return false;
    used[v] = 1;
    for (auto to : g[v]) {
        if (mt[to] == -1) {
            rev_mt[v] = to;
            mt[to] = v;
            return true;
        }
    }
    for (auto to : g[v]) {
        if (dfs(mt[to], g, mt, rev_mt, used)) {
            rev_mt[v] = to;
            mt[to] = v;
            return true;
        }
    }
    return false;
}

pair<int, vector<int>> pair_matching(int n, int m, graph<int> g
     ) {
    vector<int> mt(m, -1), used, rev_mt(n, -1);
    int cnt = 0;
    for (int it = 0; ; it++) {
        bool found = false;
        used.assign(n, 0);
        for (int i = 0; i < n; i++) {
```

```
            if (rev_mt[i] == -1 && dfs(i, g, mt, rev_mt, used))
                {
                cnt++;
                found = true;
            }
        }
        if (!found) {
            break;
        }
    }
    return {cnt, mt};
}
```

max-flow.h

Description: Maximum flow problem

Time: $\mathcal{O}\left(min(n^(2/3) * m, m^(3/2))\right)$ - all capacities are 1, $\mathcal{O}\left(n^2, m\right)$ - general

hash104 lines

```
template<typename Cap>
struct mf_graph {
    struct mf_edge {
        int from, to;
        Cap cap, flow;
        int back_id;
        int _id;
    };
    int n;
    vector<vector<mf_edge>> g;
  bool need_clear = false;

    mf_graph(int n): n(n) {
        g.assign(n, {});
    }

    void add_edge(int from, int to, Cap cap, int id = -1) {
        int id1 = g[from].size();
        int id2 = g[to].size();

        g[from].push_back(mf_edge {
            .from = from,
            .to = to,
            .cap = cap,
            .flow = Cap(),
            .back_id = id2,
            ._id = id
        });
        g[to].push_back(mf_edge {
            .from = to,
            .to = from,
            .cap = Cap(),
            .flow = Cap(),
            .back_id = id1,
            ._id = -1
        });
    }

  void clear() {
    for (int i = 0; i < n; i++) {
      for (mf_edge& e: g[i])
        e.flow = Cap();
    }
  }

    Cap flow(int s, int t, Cap limit) {
    if (need_clear)
      clear();

        vector<int> dist(n, n + 1);
        auto bfs = [&](int s, int t) {
            dist.assign(n, n + 1);
            dist[s] = 0;
            deque<int> q;
            q.push_back(s);

            while (!q.empty())
            {
                int v = q.front();
                q.pop_front();

                for (mf_edge& e : g[v]) {
                    if (e.flow < e.cap && dist[e.to] == n + 1)
                        {
                        dist[e.to] = dist[v] + 1;
                        q.push_back(e.to);
                    }
                }
            }

            return dist[t] != n + 1;
        };
        vector<int> lst(n);
        function<Cap(int, int, Cap)> dfs = [&](int v, int
             target, Cap F) {
            if (v == target)
                return F;

            Cap pushed = Cap();
            for (; lst[v] < g[v].size(); lst[v]++) {
                mf_edge& e = g[v][lst[v]];
                if (dist[e.to] == dist[v] + 1 && e.flow < e.cap
                     ) {
                    Cap x = dfs(e.to, target, min(F, e.cap - e.
                         flow));

                    if (x) {
                        e.flow += x;
                        g[e.to][e.back_id].flow -= x;
                        pushed += x;
                        F -= x;
                    }
                }
            }
            return pushed;
        };

        Cap flow = 0;
        while (bfs(s, t)) {
            lst.assign(n, 0);
            while (Cap f = dfs(s, t, limit)) {
                flow += f;
            }
        }
    need_clear = true;
        return flow;
    }
};
```

lca.h

Description: Finds lowest common ancestor of two vertices using sparce table

Time: $\mathcal{O}\left(n * log(n)\right)$ - build, $\mathcal{O}\left(1\right)$ - get

hash30 lines

```
struct LCA {
  vector<pair<int, int>> traversal;
  vector<int> pos;
  graph<int> g;
  int n;
  sparse_table<pair<int, int>> st;

  LCA(int n_, graph<int> g_, int root = 0): n(n_), g(g_) {
    pos.resize(n);
    dfs(root, -1, 0);
    st = sparse_table<pair<int, int>>(traversal.size(),
         traversal);
  }

  void dfs(int v, int pred, int depth) {
    pos[v] = traversal.size();
    traversal.push_back({depth, v});
    for (auto to : g[v]) {
      if (to != pred) {
        dfs(to, v, depth + 1);
        traversal.push_back({depth, v});
      }
    }
  }

  int get(int a, int b) {
    if (a == b)
      return a;
    return st.get(min(pos[a], pos[b]), max(pos[a], pos[b]) + 1)
         .second;
  }
};
```

two-sat.h

Description: 2-sat implementation

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

<vector> hash74 lines

```
template<typename T>
using graph = std::vector<std::vector<T>>;

struct two_sat {
    graph<int> g, rev;
    std::vector<int> used, order, comp, ans;
    int n;

    two_sat(int _n): n(_n) {
        g.assign(2 * n, {});
        rev.assign(2 * n, {});
    }

    void add_edge(int u, int v) {
        g[u].push_back(v);
        rev[v].push_back(u);
    }

    void add_clause_or(int a, bool val_a, int b, bool val_b) {
        add_edge(a + val_a * n, b + !val_b * n);
        add_edge(b + val_b * n, a + !val_a * n);
    }

    void add_clause_xor(int a, bool val_a, int b, bool val_b) {
        add_clause_or(a, val_a, b, val_b);
        add_clause_or(a, !val_a, b, !val_b);
    }

    void add_clause_and(int a, bool val_a, int b, bool val_b) {
        add_clause_xor(a, !val_a, b, val_b);
    }

    void top_sort(int v) {
        used[v] = 1;
        for (auto to : g[v]) {
            if (!used[to])
                top_sort(to);
        }
```

```
        order.push_back(v);
    }

    void compress(int v, int id) {
        comp[v] = id;
        for (auto to : rev[v]) {
            if (comp[to] == -1)
                compress(to, id);
        }
    }

    bool satisfiable() {
        order.clear();
        used.assign(2 * n, 0);
        comp.assign(2 * n, -1);
        ans.assign(n, 0);

        for (int i = 0; i < 2 * n; i++) {
            if (!used[i])
                top_sort(i);
        }
        reverse(order.begin(), order.end());
        int id = 0;
        for (auto v : order) {
            if (comp[v] == -1)
                compress(v, id++);
        }

        for (int i = 0; i < n; i++) {
            if (comp[i] == comp[i + n])
                return false;
            ans[i] = (comp[i + n] < comp[i]);
        }
        return true;
    }
};
```

eulerian-path.h

Description: Finds eulerian path and cycle for undirected graph

Time: $\mathcal{O}(m * logm)$

<vector>, <set> hash101 lines

```
using namespace std;
struct eulerian_path
{
    vector<multiset<int>>graph;
    eulerian_path(vector<vector<int>>g)
    {
        graph.resize(g.size());
        for(int i=0;i<g.size();i++)
        {
            for(int j:g[i])
            {
                graph[i].insert(j);
            }
        }
    }

    void dfs(int u, vector<int>&cycle)
    {
        auto p = graph[u];
        for(int j:p)
        {
            if(graph[u].find(j) != graph[u].end())
            {
                graph[u].erase(graph[u].find(j));
                graph[j].erase(graph[j].find(u));
                dfs(j, cycle);
            }
        }
        cycle.push_back(u);
    }

    vector<int> find_cycle(int v = 0)
    {
        for(int i = 0;i < graph.size();i++)
        {
            if(graph[i].size() % 2)
                return {};
        }
        vector<int>cycle;
        dfs(v, cycle);
        for(auto x:graph)
        {
            if(x.size())
            return {};
        }
        return cycle;
    }

    vector<int> find_path()
    {
        int st = -1, fi = -1;
        int mx = 0;
        for(int i = 0;i < graph.size();i++)
        {
            if(graph[i].size() % 2)
            {
                if(st == -1)
                    st = i;
                else
                if(fi == -1)
                    fi = i;
                else
                    return {};
            }
            if(graph[mx].size()<graph[i].size())mx = i;
        }
        if(fi == -1)
        {
            auto cycle = find_cycle(mx);
            return cycle;
        }
        graph[st].insert(fi);
        graph[fi].insert(st);
        auto cycle = find_cycle(st);
        if(!cycle.size())
        return {};
        cycle.pop_back();
        if(cycle[0]==st and cycle.back()==fi or cycle[0]==fi
             and cycle.back()==st)
        {
            return cycle;
        }
        vector<int>path;
        for(int i=0;;i++)
        {
            if(cycle[i] == st and cycle[i+1] == fi or cycle[i]
                 == fi and cycle[i+1] == st)
            {
                for(int j = i + 1;j < cycle.size();j++)
                {
                    path.push_back(cycle[j]);
                }
                for(int j = 0;j <= i;j++)
                {
                    path.push_back(cycle[j]);
                }
                break;
            }
        }
        return path;
    }

};
```

max-flow.h

Description: Maximum flow problem

Time: $\mathcal{O}\left(min(n^(2/3) * m, m^(3/2))\right)$ - all capacities are 1, $\mathcal{O}\left(n^2, m\right)$ - general

hash104 lines

```
template<typename Cap>
struct mf_graph {
    struct mf_edge {
        int from, to;
        Cap cap, flow;
        int back_id;
        int _id;
    };
    int n;
    vector<vector<mf_edge>> g;
  bool need_clear = false;

    mf_graph(int n): n(n) {
        g.assign(n, {});
    }

    void add_edge(int from, int to, Cap cap, int id = -1) {
        int id1 = g[from].size();
        int id2 = g[to].size();

        g[from].push_back(mf_edge {
            .from = from,
            .to = to,
            .cap = cap,
            .flow = Cap(),
            .back_id = id2,
            ._id = id
        });
        g[to].push_back(mf_edge {
            .from = to,
            .to = from,
            .cap = Cap(),
            .flow = Cap(),
            .back_id = id1,
            ._id = -1
        });
    }

  void clear() {
    for (int i = 0; i < n; i++) {
      for (mf_edge& e: g[i])
        e.flow = Cap();
    }
  }

    Cap flow(int s, int t, Cap limit) {
    if (need_clear)
      clear();

        vector<int> dist(n, n + 1);
        auto bfs = [&](int s, int t) {
            dist.assign(n, n + 1);
            dist[s] = 0;
            deque<int> q;
            q.push_back(s);

            while (!q.empty())
```

```
            {
                int v = q.front();
                q.pop_front();

                for (mf_edge& e : g[v]) {
                    if (e.flow < e.cap && dist[e.to] == n + 1)
                         {
                        dist[e.to] = dist[v] + 1;
                        q.push_back(e.to);
                    }
                }
            }

            return dist[t] != n + 1;
        };
        vector<int> lst(n);
        function<Cap(int, int, Cap)> dfs = [&](int v, int
             target, Cap F) {
            if (v == target)
                return F;

            Cap pushed = Cap();
            for (; lst[v] < g[v].size(); lst[v]++) {
                mf_edge& e = g[v][lst[v]];
                if (dist[e.to] == dist[v] + 1 && e.flow < e.cap
                     ) {
                    Cap x = dfs(e.to, target, min(F, e.cap - e.
                         flow));

                    if (x) {
                        e.flow += x;
                        g[e.to][e.back_id].flow -= x;
                        pushed += x;
                        F -= x;
                    }
                }
            }
            return pushed;
        };

        Cap flow = 0;
        while (bfs(s, t)) {
            lst.assign(n, 0);
            while (Cap f = dfs(s, t, limit)) {
                flow += f;
            }
        }
    need_clear = true;
        return flow;
    }
};
```

min-cut.h

Description: Start dfs from source and using edges with capacity - flow > 0, mark visited vertices. Min cut is edges between marked and unmarked vertices.
Time: $\mathcal{O}(n+m)$

min-cost-flow.h

Description: Solves minimum-cost flow problem for specific flow value (or infinity).
Time: $\mathcal{O}(F*(n+m)*log(n+m))$, where F is the amount of the flow and m is the number of added edges.
hash106 lines

```
template<typename Cap, typename Cost>
struct mcf_graph
{
    struct mcf_edge {
        int from, to;
        Cap cap, flow;
        Cost cost;
        int back_id;
        int _id;
    };
    int n;
    vector<vector<mcf_edge>> g;
    mcf_graph(int n): n(n) {
        g.assign(n, {});
    }

    void add_edge(int from, int to, Cap cap, Cost cost, int _id
         = -1) {
        int id1 = g[from].size();
        int id2 = g[to].size();
        g[from].push_back(mcf_edge{
            .from = from,
            .to = to,
            .cap = cap,
            .flow = Cap(),
            .cost = cost,
            .back_id = id2,
            ._id = _id
        });

        g[to].push_back(mcf_edge{
            .from = to,
            .to = from,
            .cap = Cap(),
            .flow = Cap(),
            .cost = -cost,
            .back_id = id1,
            ._id = -1
        });
    }

    pair<Cap, Cost> flow(int s, int t, Cap target_flow) {
        Cap flow = Cap();
        vector<pair<int, int>> pred(n);
        vector<Cost> dist(n), dual(n, 0);
        auto shortests_path = [&](int s, int t) {
            pred.assign(n, {-1, -1});
            dist.assign(n, numeric_limits<Cost>::max());
            dist[s] = Cost();
            set<pair<int, int>> q;
            q.insert({dist[s], s});

            while (!q.empty()) {
                int v = q.begin()->second;
                q.erase(q.begin());

                for (int i = 0; i < g[v].size(); i++) {
                    mcf_edge& e = g[v][i];
                    if (e.flow >= e.cap)
                        continue;

                    Cost cost = e.cost - dual[e.to] + dual[v];
                    if (dist[e.to] > dist[v] + cost) {
                        q.erase({dist[e.to], e.to});
                        dist[e.to] = dist[v] + cost;
                        pred[e.to] = {v, i};
                        q.insert({dist[e.to], e.to});
                    }
                }
            }
            if (dist[t] == numeric_limits<Cost>::max())
                return false;
            for (int v = 0; v < n; v++) {
                if (dist[v] == numeric_limits<Cost>::max())
                    continue;
                dual[v] -= dist[t] - dist[v];
            }

            return true;
        };
        Cost total_cost = {};
        while (flow < target_flow) {
            if (!shortests_path(s, t))
                break;
            Cap f = target_flow - flow;
            int cur = t;
            while (cur != s) {
                auto [p, id] = pred[cur];
                mcf_edge& e = g[p][id];
                f = min(f, e.cap - e.flow);
                cur = p;
            }
            cur = t;
            while (cur != s) {
                auto [p, id] = pred[cur];
                mcf_edge& e = g[p][id];
                e.flow += f;
                g[e.to][e.back_id].flow -= f;
                cur = p;
            }
            Cost d = -dual[s];
            flow += f;
            total_cost += f * d;
        }
        return {flow, total_cost};
    }
};
```

Strings (6)

z-function.h

Description: Z-functions, z[i] equal to the length of largest common prefix of string s and suffix of s starting at i.
Time: $\mathcal{O}(N)$, N - size of string s
hash18 lines

```
vector<int> z_function(const string& s) {
  int n = s.size();
  vector<int> z(n);
  int l = 0, r = 0;
  for(int i = 1; i < n; i++) {
    if(i < r) {
      z[i] = min(r - i, z[i - 1]);
    }
    while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
      z[i]++;
    }
    if(i + z[i] > r) {
      l = i;
      r = i + z[i];
    }
  }
  return z;
}
```

prefix-function.h

Description: The prefix function for this string is defined as an array p of length n, where p[i] is the length of the longest valid prefix of the substring s[0...i], which is also a suffix of this substring.
Time: $\mathcal{O}(N)$, N - size of string s
hash13 lines

```
vector<int> prefix_function(string s) {
```

```
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

suffix-array.h

Description: Suffix array will contain integers that represent the starting indexes of the all the suffixes of a given string, after the aforementioned suffixes are sorted.
Time: $\mathcal{O}(N * log(N))$, N - size of string s

hash65 lines

```
vector<int> suffix_arrays(string s) {
  s = s + "$";
  int n = s.size();

  std::vector<int> p(n);
  vector<vector<int>> c(20, vector<int>(n));
  int alphabet = 256;

  auto set_classes = [&](int k) {
    int classes = 0;
    c[k][p[0]] = classes++;
    for (int i = 1; i < n; i++) {
      auto cur = pair{c[k - 1][p[i]], c[k - 1][(p[i] + (1<<(k
          -1))) % n]};
      auto prev = pair{c[k - 1][p[i - 1]], c[k - 1][(p[i - 1] +
          (1<<(k-1))) % n]};
      if (cur == prev) {
        c[k][p[i]] = c[k][p[i - 1]];
      } else {
        c[k][p[i]] = classes++;
      }
    }
  };

  vector<int> cnt(alphabet);
  for (int i = 0; i < n; i++) {
    cnt[s[i]]++;
  }
  for (int i = 1; i < alphabet; i++) {
    cnt[i] += cnt[i - 1];
  }
  for (int i = n - 1; i >= 0; i--) {
    p[cnt[s[i]] - 1] = i;
    cnt[s[i]]--;

  }
  int classes = 0;
  c[0][p[0]] = classes++;
  for (int i = 1; i < n; i++) {
    if (s[p[i]] == s[p[i - 1]]) {
      c[0][p[i]] = c[0][p[i - 1]];
    } else {
      c[0][p[i]] = classes++;
    }
  }

  for (int k = 0; (1<<k) < n; k++) {
    vector<int> pn(n), cnt(n);
    for (int i = 0; i < n; i++) {
      pn[i] = (p[i] - (1<<k) + n) % n;
      cnt[c[k][pn[i]]]++;
    }

    for (int i = 1; i < n; i++)
      cnt[i] += cnt[i - 1];

    for (int i = n - 1; i >= 0; i--) {
      p[cnt[c[k][pn[i]]] - 1] = pn[i];
      cnt[c[k][pn[i]]]--;
    }

    set_classes(k + 1);
  }

  p.erase(p.begin());
  return p;
}
```

lcp-array.h

Description: Largest common prefix of substrings. Given a string s of length n, it returns the LCP array of s. Here, the LCP array of s is the array of length n-1, such that the i-th element is the length of the LCP (Longest Common Prefix) of s[sa[i]..n) and s[sa[i+1]..n).
Time: $\mathcal{O}(N)$, N - size of string s

hash31 lines

```
template <class T>
std::vector<int> lcp_array(const std::vector<T>& s,
                           const std::vector<int>& sa) {
    int n = int(s.size());
    assert(n >= 1);
    std::vector<int> rnk(n);
    for (int i = 0; i < n; i++) {
        rnk[sa[i]] = i;
    }
    std::vector<int> lcp(n - 1);
    int h = 0;
    for (int i = 0; i < n; i++) {
        if (h > 0) h--;
        if (rnk[i] == 0) continue;
        int j = sa[rnk[i] - 1];
        for (; j + h < n && i + h < n; h++) {
            if (s[j + h] != s[i + h]) break;
        }
        lcp[rnk[i] - 1] = h;
    }
    return lcp;
}

std::vector<int> lcp_array(const std::string& s, const std::
     vector<int>& sa) {
    int n = int(s.size());
    std::vector<int> s2(n);
    for (int i = 0; i < n; i++) {
        s2[i] = s[i];
    }
    return lcp_array(s2, sa);
}
```

hash.h

Description: Polynomial hashes for strings
Time: $\mathcal{O}(n * log(M))$, n - size of string s, m - module

hash47 lines

```
template<int P, int MOD>
struct hash_st {
    int n;
    vector<int> hash_, rev_, p, rev_p;
    hash_st(string s) {
        n = s.size();
        hash_.resize(n);
        rev_.resize(n);
        p.resize(n);
        rev_p.resize(n);

        p[0] = 1;
        rev_p[0] = 1;
        for (int i = 1; i < n; i++) {
            p[i] = (p[i - 1] * P) % MOD;
            rev_p[i] = fast_pow(p[i], MOD - 2, MOD);
        }

        int last = 0;
        for (int i = 0; i < n; i++) {
            hash_[i] = (last + p[i] * (s[i] - 'a' + 1)) % MOD;
            last = hash_[i];
        }
        last = 0;
        for (int i = n - 1; i >= 0; i--) {
            rev_[i] = (last + p[n - 1 - i] * (s[i] - 'a' + 1))
                 % MOD;
            last = rev_[i];
        }
    }

    int get(int l, int r) {
        r--;
        if (l == 0)
            return hash_[r];
        int x = (MOD + hash_[r] - hash_[l - 1]) % MOD;
        return (x * rev_p[l]) % MOD;
    }

    int get_rev(int l, int r) {
        r--;
        if (r == n - 1)
            return rev_[l];
        int x = (MOD + rev_[l] - rev_[r + 1]) % MOD;
        int st = n - 1 - r;
        return (x * rev_p[st]) % MOD;
    }
};
```

trie.h

Description: Trie implementation
Time: $\mathcal{O}(n)$

hash63 lines

```
template<int N = (int)1e6 + 1>
struct trie {
  struct node {
    int cnt = 0;
    array<int, 27> links;

    node() {
      links.fill(-1);
      cnt = 0;
    }
  };

  array<node, N> tree;
  int sz = 0;
  int root = 0;

  trie() {
    root = sz++;
  }

  void add(string s) {
    int cur = root;
    tree[cur].cnt++;
```

```
    for (int i = 0; i < s.size(); i++) {
      int nxt = tree[cur].links[s[i] - 'a'];
      if (nxt == -1) {
        nxt = sz++;
        tree[cur].links[s[i] - 'a'] = nxt;
      }
      tree[nxt].cnt++;
      cur = nxt;
    }
  }

  void remove(string s) {
    int cur = root;
    tree[cur].cnt--;
    for (int i = 0; i < s.size(); i++) {
      int nxt = tree[cur].links[s[i] - 'a'];
      assert(nxt != -1);
      tree[nxt].cnt--;
      cur = nxt;
    }
  }

  int get_cnt_of_str(const string& s) {
    int cur = root;
    for (int i = 0; i < s.size(); i++) {
      int nxt = tree[cur].links[s[i] - 'a'];
      if (nxt == -1) {
        return 0;
      }
      cur = nxt;
    }
    return tree[cur].cnt;
  }

  void clear() {
    for (int i = 0; i < sz; i++)
      tree[i] = node();
  }
};
```

Geometry (7)

point.h

Description: geometry formuls

<complex>, <iostream> hash124 lines

```
class point : public std::complex<long double> {
  using ld = long double;
  static constexpr long double PI = 3.141592653589793;

public:
  point() : std::complex<long double>() {}

  point(ld x, ld y) : std::complex<long double>(x, y) {}

  point(std::complex<long double> obj) : std::complex<long
      double>(obj) {}

  ld x() {
    return this->real();
  }

  ld y() {
    return this->imag();
  }

  ld x() const {
    return this->real();
  }

  ld y() const {
    return this->imag();
  }

  // a_x * b_x + a_y * b_y
  static ld dot_product(const point& a, const point& b) {
    return (conj(a) * b).real();
  }

  // a_x * b_y - a_y * b_x
  static ld cross_product(const point& a, const point& b) {
    return (conj(a) * b).imag();
  }

  static ld squared_distance(const point& a, const point& b) {
    return norm(a - b);
  }

  static ld distance(const point& a, const point& b) {
    return abs(a - b);
  }

  // angle_of_elevation of line (a, b) to oX
  static ld angle_of_elevation(const point& a, const point& b)
      {
    return arg(b - a);
  }

  // k from y = k * x + b
  static ld slope_of_line(const point& a, const point& b) {
    return tan(arg(b - a));
  }

  static point from_polar(ld r, ld theta) {
    return std::polar(r, theta);
  }

  static point rotate_above_pivot(const point& a, const ld
      theta, const point& pivot = point(0, 0)){
    return (a - pivot) * std::polar<ld>(1.0, theta) + pivot;
  }

  point& rotate(const ld theta, const point& pivot = point(0,
      0)) {
    *this = point::rotate_above_pivot(*this, theta, pivot);
    return *this;
  }

  // angle of ABC
  static ld angle(const point& a, const point& b, const point&
      c) {
    return abs(remainder(arg(a-b) - arg(c-b), 2.0 * PI));
  }

  static point project_on_vector(const point& p, const point& v
      ) {
    return v * dot_product(p, v) / norm(v);
  }

  static point project_on_line(const point& p, const point& a,
      const point& b) {
    return a + (b - a) * dot_product(p - a, b - a) / norm(b - a
        );
  }

  static point reflect_accros(const point& p, const point& a,
      const point& b) {
    return a + conj((p - a) / (b - a)) * (b - a);
  }

  // intersection of lines (a, b) and (p, q). if parallel
      returns {false, ...} else {true, intersection}.
  friend std::pair<bool, point> intersection_of_lines(const
      point& a, const point& b, const point& p, const point& q
      ) {
    ld c1 = cross_product(p - a, b - a), c2 = cross_product(q -
         a, b - a);
    if (c1 == c2) {
      return {false, {}};
    }
    return {true, (c1 * q - c2 * p) / (c1 - c2)}; // undefined
        if parallel
  }

  // returns a, b, c from a * x + b * y + c == 0 by two points
  friend std::tuple<ld, ld, ld> get_line(const point& p, const
      point& q) {
    ld a = (p.y() - q.y());
    ld b = -(p.x() - q.x());
    ld c = p.y() * (p.x() - q.x()) - p.x() * (p.y() - q.y());
    return {a, b, c};
  }

  friend ld distance_from_point_to_line(const point& p, const
      point& a, const point& b) {
    point q = project_on_line(p, a, b);

    return point::distance(p, q);
  }

  friend ld distance_from_point_to_segment(const point& p,
      const point& a, const point& b) {
    point q = project_on_line(p, a, b);
    if (std::min(a.x(), b.x()) <= q.x() && q.x() <= std::max(a.
        x(), b.x()))
      return point::distance(p, q);
    else
      return std::min(distance(p, a), distance(p, b));
  }

  friend std::istream& operator>> (std::istream& in, point& p)
      {
    ld x, y;
    in >> x >> y;
    p = point(x, y);
    return in;
  }
};
```

faces-of-planar-graph.h

Description: Finds faces of planar graph. Rreturns a vector of vertices for each face, outer face goes first. Inner faces are returned in counter-clockwise orders and the outer face is returned in clockwise order.

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

<vector> hash89 lines

```
template<typename T>
using graph = vector<vector<T>>;

struct Point {
    int64_t x, y;

    Point(int64_t x_, int64_t y_): x(x_), y(y_) {}

    Point operator - (const Point & p) const {
        return Point(x - p.x, y - p.y);
    }
```

```
    int64_t cross (const Point & p) const {
        return x * p.y - y * p.x;
    }

    int64_t cross (const Point & p, const Point & q) const {
        return (p - *this).cross(q - *this);
    }

    int half () const {
        return int(y < 0 || (y == 0 && x < 0));
    }
};

std::vector<std::vector<size_t>> find_faces(std::vector<Point>
     vertices, graph<int> adj) {
    size_t n = vertices.size();
    std::vector<std::vector<char>> used(n);
    for (size_t i = 0; i < n; i++) {
        used[i].resize(adj[i].size());
        used[i].assign(adj[i].size(), 0);
        auto compare = [&](size_t l, size_t r) {
            Point pl = vertices[l] - vertices[i];
            Point pr = vertices[r] - vertices[i];
            if (pl.half() != pr.half())
                return pl.half() < pr.half();
            return pl.cross(pr) > 0;
        };
        std::sort(adj[i].begin(), adj[i].end(), compare);
    }
    std::vector<std::vector<size_t>> faces;
    for (size_t i = 0; i < n; i++) {
        for (size_t edge_id = 0; edge_id < adj[i].size();
             edge_id++) {
            if (used[i][edge_id]) {
                continue;
            }
            std::vector<size_t> face;
            size_t v = i;
            size_t e = edge_id;
            while (!used[v][e]) {
                used[v][e] = true;
                face.push_back(v);
                size_t u = adj[v][e];
                size_t e1 = std::lower_bound(adj[u].begin(),
                     adj[u].end(), v, [&](size_t l, size_t r) {
                    Point pl = vertices[l] - vertices[u];
                    Point pr = vertices[r] - vertices[u];
                    if (pl.half() != pr.half())
                        return pl.half() < pr.half();
                    return pl.cross(pr) > 0;
                }) - adj[u].begin() + 1;
                if (e1 == adj[u].size()) {
                    e1 = 0;
                }
                v = u;
                e = e1;
            }
            std::reverse(face.begin(), face.end());
            int sign = 0;
            for (size_t j = 0; j < face.size(); j++) {
                size_t j1 = (j + 1) % face.size();
                size_t j2 = (j + 2) % face.size();
                int64_t val = vertices[face[j]].cross(vertices[
                     face[j1]], vertices[face[j2]]);
                if (val > 0) {
                    sign = 1;
                    break;
                } else if (val < 0) {
                    sign = -1;
                    break;
                }
            }
            if (sign <= 0) {
                faces.insert(faces.begin(), face);
            } else {
                faces.emplace_back(face);
            }
        }
    }
    return faces;
}
```

Misc. algorithms (8)

primes.txt

3 lines

```
p = 962592769 is such that 221 | p - 1, which may be useful.
     For hashing use 970592641
(31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-
     bit). There are 78498
primes less than 1 000 000.
```

sos-dp.h

Description: Sum over submasks
Time: $\mathcal{O}(n * 2 * n)$

hash7 lines

```
for (int i = 0; i < n; i++) {
  for (int mask = 0; mask < (1<<n); mask++) {
    if (mask&(1<<i)) {
      dp[mask] += dp[mask^(1<<i)];
    }
  }
}
```

Techniques (A)

techniques.txt

161 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
  Slope trick
  Aliens trick
  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
```