



Eidgenössische Technische Hochschule Zürich

1ETHargy

Antti Röyskö, Yuhao Yao, Marcel Bezdrighin

adapted from MIT's version of the KTH ACM Contest Template Library

2022-10-25

## Contest (1)

template.cpp	9 lines
<pre>#include "bits/stdc++.h" #define rep(i, a, n) for (auto i = a; i &lt;= (n); ++i) #define revrep(i, a, n) for (auto i = n; i &gt;= (a); --i) #define all(a) a.begin(), a.end() #define sz(a) (int) (a).size() using namespace std; using ll = long long; using pii = pair&lt;int, int&gt;; using vi = vector&lt;int&gt;;</pre>	

### 1.1 MD5 checker

hash-cpp.sh	3 lines
<pre># Hashes a cpp file, ignoring whitespace and comments. # Usage: \$ sh ./hash-cpp.sh &lt; code.cpp cpp -dD -P -fpreprocessed   tr -d '[:space:]'   md5sum</pre>	

### 1.2 Vscode config

vscode-settings.json	5 lines
<pre>{   "editor.insertSpaces": false,   "window.titleBarStyle": "custom",   "window.customMenuBarAltFocus": false, }</pre>	

Also change the following shortcuts: CopyLineDown, CopyLineUp, cursorLineEnd, cursorLineStart.

### 1.3 Notes

#### 1.3.1 Implementation Trick

Be cautious about the following:

- `__lg(0)` might cause undefined behaviour, same for `__builtin_ctz` and `__builtin_clz`.

## Misc (2)

random.cpp	6 lines
<pre>mt19937 rng(chrono::steady_clock::now().time_since_epoch().     &lt;count()); template&lt;class T&gt; T rand(T a, T b) { return uniform_int_distribution&lt;T&gt;(a, b)     &lt; (rng); } template&lt;class T&gt; T rand() { return uniform_int_distribution&lt;T&gt;()(rng); } // shuffle(perm.begin(), perm.end(), rng);</pre>	

fast-io.cpp	20 lines
<p><b>Description:</b> Fast Read for int / long long.</p> <pre>namespace fastIO {   const int BUF_SIZE = 1 &lt;&lt; 15;   char buf[BUF_SIZE], *s = buf, *t = buf;   inline char fetch() {     if (s == t) {       t = (s = buf) + fread(buf, 1, BUF_SIZE, stdin);</pre>	

<pre>    if (s == t) return EOF;   }   return *s++; }</pre> <pre>template&lt;class T&gt; inline void read(T &amp;x) {   bool sgn = 1;   T a = 0;   char c = fetch();   while (!isdigit(c)) sgn ^= (c == '-'), c = fetch();   while (isdigit(c)) a = a * 10 + (c - '0'), c = fetch();   x = sgn ? a : -a; }</pre> <pre>// hash-cpp-all = adf9f183d70e940e1930eb2081a1b271</pre>	
--	--

hilbert-mos.cpp	21 lines
<p><b>Description:</b> Hilbert curve sorting order for Mo's algorithm. Sorts queries <math>(L_i, R_i)</math> where <math>0 \leq L_i \leq R_i &lt; n</math> into order <math>\pi</math>, such that <math>\sum_i  L_{\pi_{i+1}} - L_{\pi_i}  +  R_{\pi_{i+1}} - R_{\pi_i}  = \mathcal{O}(n\sqrt{q})</math></p> <p><b>Usage:</b> hilbertOrder(n, qs) returns <math>\pi</math></p> <p><b>Time:</b> <math>\mathcal{O}(N \log N)</math>.</p> <pre>ll hilbertOrd(int y, int x, int h) {   if (h == -1) return 0;   int s = (1 &lt;&lt; h), r = (1 &lt;&lt; h) - 1;   int y0 = y &gt;&gt; h, x0 = x &gt;&gt; h;   int y1 = y &amp; r, x1 = x &amp; r;   int ny = (y0 ? y1 : (x0 ? r - x1 : x1)); // x1 : r - x1))     &lt; ;   int nx = (y0 ? x1 : (x0 ? r - y1 : y1)); // y1 : r - y1))     &lt; ; // r - y1 : y1));   return s*s * (2*x0 + (x0 ^ y0)) + hilbertOrd(ny, nx, h-1)     &lt; ; }</pre> <pre>vector&lt;int&gt; hilbertOrder(int n, const vector&lt;pair&lt;int, int&gt;     &lt; &gt;&gt;&amp; qs) {   int h = 0, q = qs.size();   while ((1 &lt;&lt; h) &lt; n) ++h;    vector&lt;pair&lt;ll, int&gt;&gt; tmp(q);   for (int i = 0; i &lt; q; ++i) tmp[i] = {hilbertOrd(qs[i].     &lt; first, qs[i].second, h - 1), i};   sort(tmp.begin(), tmp.end());    vector&lt;int&gt; res(q);   for (int qi = 0; qi &lt; q; ++qi) res[qi] = tmp[qi].second;   return res; } // hash-cpp-all = 6467dd464ea41a6009895a50f6f12523</pre>	

## Data structure (3)

fenwick.cpp	35 lines
<p><b>Description:</b> Fenwick tree with built in binary search. Can be used as a indexed set.</p> <p><b>Usage:</b> ??</p> <p><b>Time:</b> <math>\mathcal{O}(\log N)</math>.</p> <pre>class Fenwick { private:   vector&lt;ll&gt; val; public:   Fenwick(int n) : val(n+1, 0) {}    // Adds v to index i   void add(int i, ll v) {     for (++i; i &lt; val.size(); i += i &amp; -i) {</pre>	

<pre>    val[i] += v;   } }</pre> <pre>// Calculates prefix sum up to index i ll get(int i) {   ll res = 0;   for (++i; i &gt; 0; i -= i &amp; -i) {     res += val[i];   }   return res; }</pre> <pre>ll get(int a, int b) { return get(b) - get(a-1); }</pre> <pre>// Assuming prefix sums are non-decreasing, finds last     &lt; i s.t. get(i) &lt;= v int search(ll v) {   int res = 0;   for (int h = 1 &lt;&lt; 30; h; h &gt;= 1) {     if ((res   h) &lt; val.size() &amp;&amp; val[res   h] &lt;= v) {       res  = h;       v -= val[res];     }   }   return res - 1; }</pre> <pre>//; // hash-cpp-all = 0d390772acaff4360d0f4d76da45148e</pre>	
--	--

segtree.cpp	58 lines
<p><b>Description:</b> Segment tree supporting range addition and range sum, minimum queries</p> <p><b>Usage:</b> ??</p> <p><b>Time:</b> <math>\mathcal{O}(\log N)</math>.</p> <pre>// Segment tree for range addition, range sum and range     &lt; minimum. class SegTree { private:   vector&lt;ll&gt; sum, minv, tag;   int h = 1;    // Returns length of interval corresponding to position     &lt; i   ll len(int i) { return h &gt;&gt; (31 - __builtin_clz(i)); }</pre> <pre>void apply(int i, ll v) {   sum[i] += v * len(i);   minv[i] += v;   if (i &lt; h) tag[i] += v; }</pre> <pre>void push(int i) {   if (tag[i] == 0) return;   apply(2*i, tag[i]);   apply(2*i+1, tag[i]);   tag[i] = 0; }</pre>	

<pre>ll recGetSum(int a, int b, int i, int ia, int ib) {   if (ib &lt;= a    b &lt;= ia) return 0;   if (a &lt;= ia &amp;&amp; ib &lt;= b) return sum[i];   push(i);   int im = (ia + ib) &gt;&gt; 1;   return recGetSum(a, b, 2*i, ia, im) + recGetSum(a, b,     &lt; 2*i+1, im, ib); }</pre> <pre>ll recGetMin(int a, int b, int i, int ia, int ib) {   if (ib &lt;= a    b &lt;= ia) return 4 * (ll)1e18;</pre>	
--	--

```

    if (a <= ia && ib <= b) return minv[i];
    push(i);
    int im = (ia + ib) >> 1;
    return min(recGetMin(a, b, 2*i, ia, im), recGetMin(a,
        ↪ b, 2*i+1, im, ib));
}
void recApply(int a, int b, ll v, int i, int ia, int ib
    ↪) {
    if (ib <= a || b <= ia) return;
    if (a <= ia && ib <= b) apply(i, v);
    else {
        push(i);
        int im = (ia + ib) >> 1;
        recApply(a, b, v, 2*i, ia, im);
        recApply(a, b, v, 2*i+1, im, ib);
        sum[i] = sum[2*i] + sum[2*i+1];
        minv[i] = min(minv[2*i], minv[2*i+1]);
    }
}
public:
SegTree(int n) {
    while(h < n) h *= 2;
    sum.resize(2*h, 0);
    minv.resize(2*h, 0);
    tag.resize(h, 0);
}
ll rangeSum(int a, int b) { return recGetSum(a, b+1, 1,
    ↪ 0, h); }
ll rangeMin(int a, int b) { return recGetMin(a, b+1, 1,
    ↪ 0, h); }
void rangeAdd(int a, int b, ll v) { recApply(a, b+1, v,
    ↪ 1, 0, h); }
}; // hash-cpp-all = e3e31721068f2f6661b4302da9d50cb9

```

## rmq.cpp

**Description:** range minimum query data structure with low memory and fast queries

**Usage:** ??

**Time:**  $\mathcal{O}(N)$  preprocessing,  $\mathcal{O}(1)$  query.

63 lines

```

int firstBit(ull x) { return __builtin_ctzll(x); }
int lastBit(ull x) { return 63 - __builtin_clzll(x); }

```

//  $\mathcal{O}(n)$  preprocessing,  $\mathcal{O}(1)$  RMQ data structure.

```

template<class T>
class RMQ {
private:
    const int H = 6; // Block size is  $2^H$ 
    const int B = 1 << H;
    vector<T> vec; // Original values
    vector<ull> mins; // Min bits
    vector<int> tbl; // sparse table
    int n, m;

    // Get index with minimum value in range [a, a + len)
    ↪ for  $0 \leq len \leq B$ 
    int getShort(int a, int len) const {
        return a + lastBit(mins[a] & (~1ull >> (64 - len)));
    }
    int minInd(int ia, int ib) const {
        return vec[ia] < vec[ib] ? ia : ib;
    }
public:
    RMQ(const vector<T>& vec_) : vec(vec_), mins(vec_.size
        ↪) {
        n = vec.size();
        m = (n + B - 1) >> H;
    }

```

```

// Build sparse table
int h = lastBit(m) + 1;
tbl.resize(h*m);
for (int j = 0; j < m; ++j) tbl[j] = j << H;
for (int i = 0; i < n; ++i) tbl[i >> H] = minInd(tbl[
    ↪ i >> H], i);
for (int j = 1; j < h; ++j) {
    for (int i = j*m; i < (j+1)*m; ++i) {
        int i2 = min(i + (1 << (j-1)), (j+1)*m - 1);
        tbl[i] = minInd(tbl[i-m], tbl[i2-m]);
    }
}
// Build min bits
ull cur = 0;
for (int i = n-1; i >= 0; --i) {
    for (cur <= 1; cur > 0; cur ^= cur & -cur) {
        if (vec[i + firstBit(cur)] < vec[i]) break;
    }
    cur |= 1;
    mins[i] = cur;
}
int argmin(int a, int b) const {
    ++b; // to make the range inclusive
    int len = min(b-a, B);
    int ind1 = minInd(getShort(a, len), getShort(b-len,
        ↪ len));

    int ax = (a >> H) + 1;
    int bx = (b >> H);
    if (ax >= bx) return ind1;
    else {
        int h = lastBit(bx-ax);
        int ind2 = minInd(tbl[h*m + ax], tbl[h*m + bx - (1
            ↪ << h)));
        return minInd(ind1, ind2);
    }
}
int get(int a, int b) const { return vec[argmin(a, b)];
    ↪ }
}; // hash-cpp-all = 3dd48eb5fa928d12b0e5b263ce842625

```

## cartesian-tree.cpp

**Description:** Cartesian Tree of array *as* (of distinct values) of length *N*. Node with smaller depth has smaller value. Set *gr* = 1 to have top with the greatest value. Returns the root of Cartesian Tree, left sons of nodes and right sons of nodes. (-1 means no left son / right son.)

**Time:**  $\mathcal{O}(N)$  for construction.

14 lines

```

template<class T>
auto CartesianTree(const vector<T> &as, int gr = 0) {
    int n = sz(as);
    vi ls(n, -1), rs(n, -1), sta;
    rep(i, 0, n - 1) {
        while (sz(sta) && ((as[i] < as[sta.back()]) ^ gr)) {
            ls[i] = sta.back();
            sta.pop_back();
        }
        if (sz(sta)) rs[sta.back()] = i;
        sta.push_back(i);
    }
    return make_tuple(sta[0], ls, rs);
} // hash-cpp-all = 45ac593851f901756dd697a39dbbc90f

```

## sparse-table.cpp

**Description:** Sparse Table of an array of length *N*.

**Time:**  $\mathcal{O}(N \log N)$  for construction,  $\mathcal{O}(1)$  per query.

19 lines

```

template<class T, class F = function<T(const T&, const T&)
    ↪ >>>
class SparseTable {
    int n;
    vector<vector<T>> st;
    const F func;
public:
    SparseTable(const vector<T> &init, const F &f): n(sz(init
        ↪)), func(f) {
        assert(n > 0);
        st.assign(__lg(n) + 1, vector<T>(n));
        st[0] = init;
        rep(i, 1, __lg(n)) rep(x, 0, n - (1 << i)) st[i][x] =
            ↪ func(st[i - 1][x], st[i - 1][x + (1 << (i - 1))]);
    }

    T ask(int l, int r) {
        assert(0 <= l && l <= r && r < n);
        int k = __lg(r - l + 1);
        return func(st[k][l], st[k][r - (1 << k) + 1]);
    }
}; // hash-cpp-all = ba1bdd7413e0da2668e14467f92cf02d

```

## sparse-table-2d.cpp

**Description:** 2D Sparse Table of 2D vector of size  $N \times M$ .

**Time:**  $\mathcal{O}(NM \log N \log M)$  for construction,  $\mathcal{O}(1)$  per query.

37 lines

```

template<class T, class F = function<T(const T&, const T&)
    ↪ >>>
class SparseTable2D {
    using vt = vector<T>;
    using vvt = vector<vt>;

    int n, m;
    vector<vector<vvt>> st;
    const F func;
public:
    SparseTable2D(const vvt &init, const F &f): n(sz(init)),
        ↪ func(f) {
        assert(n > 0);
        m = sz(init[0]);
        assert(m > 0);

        st.assign(__lg(n) + 1, vector<vvt>(__lg(m) + 1, vvt(n,
            ↪ vt(m))));
        st[0][0] = init;
        rep(j, 1, __lg(m)) rep(x, 0, n - 1) rep(y, 0, m - (1 <<
            ↪ j)) {
            st[0][j][x][y] = func(st[0][j - 1][x][y], st[0][j -
                ↪ 1][x][y + (1 << (j - 1))]);
        }
        rep(i, 1, __lg(n)) rep(j, 0, __lg(m)) rep(x, 0, n - (1
            ↪ << i)) rep(y, 0, m - (1 << j)) {
            st[i][j][x][y] = func(st[i - 1][j][x][y], st[i - 1][j
                ↪ << i][x + (1 << (i - 1))][y]);
        }
    }
}

```

```

T ask(int x1, int y1, int x2, int y2) {
    assert(0 <= x1 && x1 <= x2 && x2 < n);
    assert(0 <= y1 && y1 <= y2 && y2 < m);
    int kx = __lg(x2 - x1 + 1);
    int ky = __lg(y2 - y1 + 1);
    int lx = 1 << kx;

```

```

int ly = 1 << ky;
T res = func(st[kx][ky][x1][y1], st[kx][ky][x1][y2 - ly
    ↪ + 1]);
res = func(res, st[kx][ky][x2 - lx + 1][y1]);
res = func(res, st[kx][ky][x2 - lx + 1][y2 - ly + 1]);
return res;
}
}; // hash-cpp-all = 3da0c2d78858b5b3c198f4757545f121

```

### lichao.cpp

**Description:** Li Chao tree. Given x-coordinates, supports adding lines and computing minimum Y-coordinate at a given input x-coordinate

**Usage:** ??

**Time:**  $\mathcal{O}(\log N)$ .

39 lines

```

struct Line {
    ll a, b;
    ll eval(ll x) const { return a*x + b; }
};
class LiChao {
private:
    const static ll INF = 4e18;
    vector<Line> tree; // Tree of lines
    vector<ll> xs; // x-coordinate of point i
    int k = 1; // Log-depth of the tree

    int mapInd(int j) const {
        int z = __builtin_ctz(j);
        return ((1<<(k-z)) | (j>>z)) >> 1;
    }
    bool comp(const Line& a, int i, int j) const {
        return a.eval(xs[j]) < tree[i].eval(xs[j]);
    }
public:
    LiChao(const vector<ll>& points) {
        while(points.size() >> k) ++k;
        tree.resize(1 << k, {0, INF});
        xs.resize(1 << k, points.back());
        for (int i = 0; i < points.size(); ++i) xs[mapInd(i
            ↪ +1)] = points[i];
    }
    void addLine(Line line) {
        for (int i = 1; i < tree.size(); i) {
            if (comp(line, i, i)) swap(line, tree[i]);
            if (line.a > tree[i].a) i = 2*i;
            else i = 2*i+1;
        }
    }
    ll minVal(int j) const {
        j = mapInd(j+1);
        ll res = INF;
        for (int i = j; i > 0; i /= 2) res = min(res, tree[i
            ↪ ].eval(xs[j]));
        return res;
    }
}; // hash-cpp-all = 51ad9045bfff4d74f5c7b851530e02304

```

### skew-heap.cpp

**Description:** Skew heap: a priority queue with fast merging

**Usage:** ??

**Time:** all operations  $\mathcal{O}(\log N)$ .

38 lines

```

// Skew Heap
class SkewHeap {
private:
    struct Node {
        ll val, inc = 0;

```

```

int ch[2] = {-1, -1};
Node(ll v) : val(v) {}
};
vector<Node> nodes;
public:
    int makeNode(ll v) {
        nodes.emplace_back(v);
        return (int)nodes.size() - 1;
    }

    // Increment all values in heap p by v
    void add(int i, ll v) {
        if (i == -1) return;
        nodes[i].val += v;
        nodes[i].inc += v;
    }

    // Merge heaps a and b
    int merge(int a, int b) {
        if (a == -1 || b == -1) return a + b + 1;
        if (nodes[a].val > nodes[b].val) swap(a, b);
        if (nodes[a].inc) {
            add(nodes[a].ch[0], nodes[a].inc);
            add(nodes[a].ch[1], nodes[a].inc);
            nodes[a].inc = 0;
        }
        swap(nodes[a].ch[0], nodes[a].ch[1]);
        nodes[a].ch[0] = merge(nodes[a].ch[0], b);
        return a;
    }
    pair<int, ll> top(int i) const { return {i, nodes[i].
        ↪ val}; }
    void pop(int& p) { p = merge(nodes[p].ch[0], nodes[p].
        ↪ ch[1]); }
}; // hash-cpp-all = c72cc101090bd3027c2442ee11cee862

```

### fast-priqueue.cpp

**Description:** Struct for priority queue operations on index set  $[0, n-1]$ .

**Usage:** push(i, v) overwrites value at position i if one already exists. decKey is faster, but does nothing if the new key is smaller than the old one. top and pop can segfault if called on an empty priority queue.

**Time:**  $\mathcal{O}(\log N)$ .

22 lines

```

struct Prique {
    const ll INF = 4 * (ll)1e18;
    vector<pair<ll, int>> data;
    const int n;

    Prique(int siz) : n(siz), data(2*siz, {INF, -1}) { data
        ↪ [0] = {-INF, -1}; }
    bool empty() const { return data[1].second >= INF; }
    pair<ll, int> top() const { return data[1]; }

    void push(int i, ll v) {
        data[i+n] = {v, (v >= INF ? -1 : i)};
        for (i += n; i > 1; i >= 1) data[i>>1] = min(data[i],
            ↪ data[i^1]);
    }
    void decKey(int i, ll v) {
        for (int j = i+n; data[j].first > v; j >= 1) data[j] =
            ↪ {v, i};
    }
    pair<ll, int> pop() {
        auto res = data[1];
        push(res.second, INF);
        return res;
    }

```

```

}
}; // hash-cpp-all = 08f397034ba143af3dc3c98b96f9a634

```

### persistent-segtree.cpp

**Description:** Persistent Segment Tree of range  $[0, N-1]$ . Point apply and thus no lazy propagation. Always define a global *apply* function to tell segment tree how you apply modification. Combine is set as + operation. If you use your own struct, then please define constructor and + operation. In constructor, q is the number of *pointApply* you will use.

**Usage:** Point Add and Range Sum.

void apply(int &a, int b) { a += b; } // global

...

PersistSegtree<int> pseg(10, 1); // len = 10 and 1 update.

int rt = 0; // empty node.

int newrt = pseg.pointApply(rt, 9, 1); // add 1 to last

position (position 9).

int sum = pseg.rangeAsk(newrt, 7, 9); // ask the sum

between position 7 and 9, wrt version newrt.

**Time:**  $\mathcal{O}(\log N)$  per operation.

62 lines

```

template<class Info> struct PersistSegtree {
    struct node { Info info; int ls, rs; }; // hash-cpp-1
    int n;
    vector<node> t;
    // node 0 is left as virtual empty node.
    PersistSegtree(int n, int q): n(n), t(1) {
        assert(n > 0);
        t.reserve(q * (__lg(n) + 2) + 1);
    }

```

// pointApply returns the id of new root.

template<class... T>

```

int pointApply(int rt, int pos, const T&... val) {
    auto dfs = [&](auto &dfs, int &i, int l, int r) {
        t.push_back(t[i]);
        i = sz(t) - 1;

```

```

        if (l == r) {
            ::apply(t[i].info, val...);
            return;
        }
        int mid = (l + r) >> 1;
        if (pos <= mid) dfs(dfs, t[i].ls, l, mid);
        else dfs(dfs, t[i].rs, mid + 1, r);
        t[i].info = t[t[i].ls].info + t[t[i].rs].info;
    };
    dfs(dfs, rt, 0, n - 1);
    return rt;
}

```

```

Info rangeAsk(int rt, int ql, int qr) {
    Info res{};
    auto dfs = [&](auto &dfs, int i, int l, int r) {
        if (i == 0 || qr < l || r < ql) return;
        if (ql <= l && r <= qr) {
            res = res + t[i].info;
            return;
        }
        int mid = (l + r) >> 1;
        dfs(dfs, t[i].ls, l, mid);
        dfs(dfs, t[i].rs, mid + 1, r);
    };
    dfs(dfs, rt, 0, n - 1);
    return res;
} // hash-cpp-1 = 9569f9abfb3ee296b5ea10a5f70b8ddb

```

```
// lower_bound on prefix sums of difference between two
// versions.
int lower_bound(int rt_l, int rt_r, Info val) { // hash-
// cpp-2
Info sum{};
auto dfs = [&](auto &dfs, int x, int y, int l, int r) {
    if (l == r) return sum + t[y].info - t[x].info >= val
        ? l : l + 1;
    int mid = (l + r) >> 1;
    Info s = t[t[y].ls].info - t[t[x].ls].info;
    if (sum + s >= val) return dfs(dfs, t[x].ls, t[y].ls,
        l, mid);
    else {
        sum = sum + s;
        return dfs(dfs, t[x].rx, t[y].rs, mid + 1, r);
    }
};
return dfs(dfs, rt_l, rt_r, 0, n - 1);
} // hash-cpp-2 = 8a719a17e052e3651546ac8d8a122c9c
};
```

### segtree-2d.cpp

**Description:** 2D Segment Tree of range  $[oL, oR] \times [iL, iR]$ . Point apply and thus no lazy propagation. Always define a global *apply* function to tell segment tree how you apply modification. Combine is set as + operation. If you use your own struct, then please define constructor and + operation. In constructor, *q* is the number of *pointApply* you will use. *oL*, *oR*, Note that range parameters can be negative.

**Usage:** Point Add and Range (Rectangle) Sum.

void apply(int &a, int b) { a += b; } // global

```
...
SegTree2D<int> pseg(-5, 5, -5, 5, 1); // [-5, 5] * [-5, 5]
and 1 update.
int rt = 0; // empty node.
rt = pseg.pointApply(rt, 2, -1, 1); // add 1 to position
(2, -1).
int sum = pseg.rangeAsk(rt, 3, 4, -2, -1); // ask the sum
in rectangle [3, 4] * [-2, -1].
```

**Time:**  $\mathcal{O}(\log(oR - oL + 1) \times \log(iR - iL + 1))$  per operation. 74 lines

```
template<class Info> struct SegTree2D {
    struct iNode { Info info; int ls, rs; };
    struct oNode { int id; int ls, rs; };

    int oL, oR, iL, iR;
    // change to array to accelerate, since allocating takes
    // time. (saves ~ 200ms when allocating 1e7)
    vector<iNode> it;
    vector<oNode> ot;

    // node 0 is left as virtual empty node.
    SegTree2D(int oL, int oR, int iL, int iR, int q): oL(oL),
        oR(oR), iL(iL), iR(iR), it(1), ot(1) {
        it.reserve(q * (__lg(oR - oL + 1) + 2) * (__lg(iR - iL
            + 1) + 2) + 1);
        ot.reserve(q * (__lg(oR - oL + 1) + 2) + 1);
    }
};
```

// return new root id.

```
template<class... T>
int pointApply(int rt, int op, int ip, const T&... val) {
    auto idfs = [&](auto &dfs, int &i, int l, int r) {
        if (!i) {
            it.push_back({});
            i = sz(it) - 1;
        }
        if (l == r) {
```

```
::apply(it[i].info, val...);
        return;
    }
    int mid = (l + r) >> 1;
    auto &[info, ls, rs] = it[i];
    if (ip <= mid) dfs(dfs, ls, l, mid);
    else dfs(dfs, rs, mid + 1, r);
    info = it[ls].info + it[rs].info;
};
auto odfs = [&](auto &dfs, int &i, int l, int r) {
    if (!i) {
        ot.push_back({});
        i = sz(ot) - 1;
    }
    idfs(idfs, ot[i].id, iL, iR);
    if (l == r) return;
    int mid = (l + r) >> 1;
    if (op <= mid) dfs(dfs, ot[i].ls, l, mid);
    else dfs(dfs, ot[i].rs, mid + 1, r);
};
odfs(odfs, rt, oL, oR);
return rt;
};

Info rangeAsk(int rt, int qol, int qor, int qil, int qir)
// {
Info res{};
auto idfs = [&](auto &dfs, int i, int l, int r) {
    if (!i || qir < l || r < qil) return;
    if (qil <= l && r <= qir) {
        res = res + it[i].info;
        return;
    }
    int mid = (l + r) >> 1;
    dfs(dfs, it[i].ls, l, mid);
    dfs(dfs, it[i].rs, mid + 1, r);
};
auto odfs = [&](auto &dfs, int i, int l, int r) {
    if (!i || qor < l || r < qol) return;
    if (qol <= l && r <= qor) {
        idfs(idfs, ot[i].id, iL, iR);
        return;
    }
    int mid = (l + r) >> 1;
    dfs(dfs, ot[i].ls, l, mid);
    dfs(dfs, ot[i].rs, mid + 1, r);
};
odfs(odfs, rt, oL, oR);
return res;
};
}; // hash-cpp-all = abc3c0ce75b1b8cfcc9b974e0b8cfdfa
```

### treap.cpp

**Description:** A Treap with lazy tag support. Default behaviour supports join, split, reverse and sum.

**Time:** All updates are  $\mathcal{O}(\log N)$

60 lines

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
// count());
int rand() { return uniform_int_distribution<int>{}(rng); }
```

```
struct Treap {
    private:
        const int pri;
        Treap *le = 0, *ri = 0;
        ll val, sum;
        int siz = 1, flip = 0;
```

```
void update() {
    siz = 1 + getSiz(le) + getSiz(ri);
    sum = val + getSum(le) + getSum(ri);
}

void push() {
    if (flip) {
        swap(le, ri);
        reverse(le);
        reverse(ri);
        flip = 0;
    }
}

public:
    Treap(ll v) : val(v), sum(v), pri(rand()) {}
    ~Treap() { delete le; delete ri; }

    static int getSiz(Treap* x) { return x ? x->siz : 0; }
    static ll getSum(Treap* x) { return x ? x->sum : 0; }
    static void reverse(Treap* x) { if (x) x->flip ^= 1; }

    static Treap* join(Treap* a, Treap* b) {
        if (!a || !b) return a ? a : b;
        Treap* res = (a->pri < b->pri ? a : b);

        res->push();
        if (res == a) a->ri = join(a->ri, b);
        else b->le = join(a, b->le);
        res->update();
        return res;
    }

    // Split the treap into a left and right part, the left
    // of size "le_siz"
    static pair<Treap*, Treap*> split(Treap* x, int le_siz)
    {
        if (!le_siz || !x) return {0, x};
        x->push();

        Treap *oth;
        int rem = le_siz - getSiz(x->le) - 1;
        if (rem < 0) {
            tie(oth, x->le) = split(x->le, le_siz);
            x->update();
            return {oth, x};
        } else {
            tie(x->ri, oth) = split(x->ri, rem);
            x->update();
            return {x, oth};
        }
    }
};

}; // hash-cpp-all = 4f72bba8689af456118ff9f9c60d6c6f6
```

### pq-tree.cpp

1 lines

// TODO

### matrix-seg.cpp

1 lines

// TODO: segment tree for historic information

## 3.1 PBDS

### pbds-hash-map.cpp

5 lines

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/hash_policy.hpp>
```

```
using namespace __gnu_pbds;
template<class A, class B>
using HashMap = gp_hash_table<A, B>;
```

### pbds-leftist-tree.cpp

5 lines

```
#include<ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;
template<class T>
using Heap = __gnu_pbds::priority_queue<T, greater<T>,
    ↳binomial_heap_tag>; // smallest value at the top.
// Use $a.join(b)$ to merge heap $b$ to heap $a$. After
    ↳merging, $b$ will be empty.
```

### pbds-ordered-set.cpp

7 lines

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template<class T>
using Oset = tree<T, null_type, less<T>, rb_tree_tag,
    ↳tree_order_statistics_node_update>; // if null_type
    ↳does not work, then use null_mapped_type instead.
// order_of_key(x) returns the number of elements which are
    ↳ smaller than x. (quite like lower_bound.)
// find_by_order(x) returns the x-th smallest element.
```

## Graph algorithms (4)

### 4.1 Flows

#### dinic.cpp

**Description:** Dinic algorithm for flow graph  $G = (V, E)$ . You can get a minimum  $src - sink$  cut easily. To get such minimum cut, first run  $MaxFlow(src, sink)$ . Then you can run  $getMinCut()$  to obtain a Minimum Cut (vertices in the same part as  $src$  are returned).

**Time:**  $\mathcal{O}(|V|^2|E|)$  for arbitrary networks.  $\mathcal{O}(|E|\sqrt{|V|})$  for bipartite/unit network.  $\mathcal{O}(\min|V|^{2/3}, |E|^{1/2}|E|)$  for networks with only unit capacities.

72 lines

```
template<class Cap = int, Cap Cap_MAX = numeric_limits<Cap>
    ↳::max()>
struct Dinic {
    int n; // hash-cpp-1
    struct E { int to; Cap a; }; // Endpoint & Admissible
        ↳flow.
    vector<E> es;
    vector<vi> g;
    vi dis; // Put it here to get the minimum cut easily.
```

```
Dinic(int n): n(n), g(n) {}
```

```
void addEdge(int u, int v, Cap c, bool dir = 1) {
    g[u].push_back(sz(es)); es.push_back({v, c});
    g[v].push_back(sz(es)); es.push_back({u, dir ? 0 : c});
}
```

```
Cap MaxFlow(int src, int sink) {
    auto revbfs = [&]() {
        dis.assign(n, -1);
        dis[sink] = 0;
        vi que{sink};
```

```
    rep(ind, 0, sz(que) - 1) {
        int now = que[ind];
```

```
        for (auto i: g[now]) {
            int v = es[i].to;
            if (es[i ^ 1].a > 0 && dis[v] == -1) {
                dis[v] = dis[now] + 1;
                que.push_back(v);
                if (v == src) return 1;
            }
        }
    }
    return 0;
};

vi cur;
auto dfs = [&](auto &dfs, int now, Cap flow) {
    if (now == sink) return flow;
    Cap res = 0;
    for (int &ind = cur[now]; ind < sz(g[now]); ind++) {
        int i = g[now][ind];
        auto [v, c] = es[i];
        if (c > 0 && dis[v] == dis[now] - 1) {
            Cap x = dfs(dfs, v, min(flow - res, c));
            res += x;
            es[i].a -= x;
            es[i ^ 1].a += x;
        }
        if (res == flow) break;
    }
    return res;
};

Cap ans = 0;
while (revbfs()) {
    cur.assign(n, 0);
    ans += dfs(dfs, src, Cap_MAX);
}
return ans;
} // hash-cpp-1 = 0099c35a07ab0465ecf3ddb9b105db6f
```

```
// Returns a min-cut containing the src.
vi getMinCut() { // hash-cpp-2
    vi res;
    rep(i, 0, n - 1) if (dis[i] == -1) res.push_back(i);
    return res;
} // hash-cpp-2 = f8bc377d2af3ac0d3b75bbac2e4f7e9

// Gives flow on edge assuming it is directed/undirected.
    ↳ Undirected flow is signed.
Cap getDirFlow(int i) { return es[i * 2 + 1].a; }
Cap getUndirFlow(int i) { return (es[i * 2 + 1].a - es[i
    ↳ * 2].a) / 2; }
};
```

### costflow-successive-shortest-path.cpp

**Description:** Successive Shortest Path for flow graph  $G = (V, E)$ . Run  $mincostflow(src, sink)$  for some  $src$  and  $sink$  to get the minimum cost and the maximum flow. For negative costs, Bellman-Ford is necessary. **Time:**  $\mathcal{O}(|F||E|\log|E|)$  for non-negative costs, where  $|F|$  is the size of maximum flow.  $\mathcal{O}(|V||E| + |F||E|\log|E|)$  for arbitrary costs.

61 lines

```
template<class Cap, class Cost, Cap Cap_MAX =
    ↳numeric_limits<Cap>::max(), Cost Cost_MAX =
    ↳numeric_limits<Cost>::max() / 4>
struct SuccessiveShortestPath {
    int n;
    struct E { int to; Cap a; Cost w; };
    vector<E> es;
```

```
vector<vi> g;
vector<Cost> h;
```

```
SuccessiveShortestPath(int n): n(n), g(n), h(n) {}
```

```
void addEdge(int u, int v, Cap c, Cost w) {
    g[u].push_back(sz(es)); es.push_back({v, c, w});
    g[v].push_back(sz(es)); es.push_back({u, 0, -w});
}
```

```
pair<Cost, Cap> mincostflow(int src, int sink, Cap
    ↳mx_flow = Cap_MAX) {
    // Run Bellman-Ford first if necessary.
    h.assign(n, Cost_MAX);
    h[src] = 0;
    rep(rd, 1, n) rep(now, 0, n - 1) for (auto i: g[now]) {
        auto [v, c, w] = es[i];
        if (c > 0) h[v] = min(h[v], h[now] + w);
    }
    // Bellman-Ford stops here.
```

```
Cost cost = 0;
Cap flow = 0;
while (mx_flow) {
    priority_queue<pair<Cost, int>> pq;
    vector<Cost> dis(n, Cost_MAX);
    dis[src] = 0; pq.emplace(0, src);
```

```
    vi pre(n, -1), mark(n, 0);
    while (sz(pq)) {
        auto [d, now] = pq.top(); pq.pop();
        // Using mark[] is safer than compare -d and dis[
            ↳now] when the Cost = double.
        if (mark[now]) continue;
        mark[now] = 1;
        for (auto i: g[now]) {
            auto [v, c, w] = es[i];
            Cost off = dis[now] + w + h[now] - h[v];
            if (c > 0 && dis[v] > off) {
                dis[v] = off;
                pq.emplace(-dis[v], v);
                pre[v] = i;
            }
        }
    }
    if (pre[sink] == -1) break;
```

```
    rep(i, 0, n - 1) if (dis[i] != Cost_MAX) h[i] += dis[
        ↳i];
    Cap aug = mx_flow;
    for (int i = pre[sink]; ~i; i = pre[es[i ^ 1].to])
        ↳aug = min(aug, es[i].a);
    for (int i = pre[sink]; ~i; i = pre[es[i ^ 1].to]) es
        ↳[i].a -= aug, es[i ^ 1].a += aug;
    mx_flow -= aug;
    flow += aug;
    cost += aug * h[sink];
}
return {cost, flow};
}
} // hash-cpp-all = 2f6de2add5c8caaf0940e67ca83c82aa
```

### 4.2 Matchings



## kuhn-matching.cpp

**Description:** Kuhn Matching algorithm for **bipartite** graph  $G = (L \cup R, E)$ . Edges  $E$  should be described as pairs such that pair  $(x, y)$  means that there is an edge between the  $x$ -th vertex in  $L$  and the  $y$ -th vertex in  $R$ . Returns a vector  $lm$ , where  $lm[i]$  denotes the vertex in  $R$  matched to the  $i$ -th vertex in  $R$ .

**Time:**  $\mathcal{O}((|L| + |R|)|E|)$ .

22 lines

```
vi Kuhn(int n, int m, const vector<pii> &es) {
    vector<vi> g(n);
    for (auto [x, y]: es) g[x].push_back(y);
    vi rm(m, -1);
    rep(i, 0, n - 1) {
        vi vis(m);
        auto dfs = [&](auto &dfs, int x) -> int {
            for (auto y: g[x]) if (vis[y] == 0) {
                vis[y] = 1;
                if (rm[y] == -1 || dfs(dfs, rm[y])) {
                    rm[y] = x;
                    return 1;
                }
            }
            return 0;
        };
        dfs(dfs, i);
    }
    vi lm(n, -1);
    rep(i, 0, m - 1) if (rm[i] != -1) lm[rm[i]] = i;
    return lm;
} // hash-cpp-all = 799e88c72327efb98bd13f428b7ee8db
```

## hopcroft.cpp

**Description:** Fast bipartite matching for **bipartite** graph  $G = (L \cup R, E)$ . Edges  $E$  should be described as pairs such that pair  $(x, y)$  means that there is an edge between the  $x$ -th vertex in  $L$  and the  $y$ -th vertex in  $R$ . You can also get a vertex cover of a bipartite graph easily.

**Time:**  $\mathcal{O}(|E|\sqrt{|L| + |R|})$ .

56 lines

```
struct Hopcroft {
    int L, R; // hash-cpp-1
    vi lm, rm; // record the matched vertex for each vertex
                // on both sides.
    vi ldis, rdis; // put it here so you can get vertex cover
                // easily.

    Hopcroft(int L, int R, const vector<pii> &es): L(L), R(R)
        //, lm(L, -1), rm(R, -1) {
        vector<vi> g(L);
        for (auto [x, y]: es) g[x].push_back(y);

        while (1) {
            ldis.assign(L, -1);
            rdis.assign(R, -1);
            bool ok = 0;
            vi que;
            rep(i, 0, L - 1) if (lm[i] == -1) {
                que.push_back(i);
                ldis[i] = 0;
            }
            rep(ind, 0, sz(que) - 1) {
                int i = que[ind];
                for (auto j: g[i]) if (rdis[j] == -1) {
                    rdis[j] = ldis[i] + 1;
                    if (rm[j] != -1) {
                        ldis[rm[j]] = rdis[j] + 1;
                        que.push_back(rm[j]);
                    } else ok = 1;
                }
            }
        }
    }
};
```

```
    }
}

if (ok == 0) break;
vi vis(R); // changing to static does not speed up.

auto find = [&](auto &dfs, int i) -> int {
    for (auto j: g[i]) if (vis[j] == 0 && rdis[j] ==
        // ldis[i] + 1) {
        vis[j] = 1;
        if (rm[j] == -1 || dfs(dfs, rm[j])) {
            lm[i] = j;
            rm[j] = i;
            return 1;
        }
    }
    return 0;
};
rep(i, 0, L - 1) if (lm[i] == -1) find(find, i);
} // hash-cpp-1 = 1bdeb27ebf133b92ed0dac89528c768e

vi getMatch() { return lm; } // returns lm.

pair<vi, vi> vertex_cover() { // hash-cpp-2
    vi lvc, rvc;
    rep(i, 0, L - 1) if (ldis[i] == -1) lvc.push_back(i);
    rep(j, 0, R - 1) if (rdis[j] != -1) rvc.push_back(j);
    return {lvc, rvc};
} // hash-cpp-2 = 4cfcc7973485543721e0bf5f6f67e3ce
};
```

## blossom.cpp

**Description:** Maximum matching of a **general** graph  $G = (V, E)$ . Edges  $E$  should be described as pairs such that pair  $(u, v)$  means that there is an edge between vertex  $u$  and vertex  $v$ .

**Time:**  $\mathcal{O}(|V||E|)$ .

81 lines

```
vi Blossom(int n, const vector<pii> &es) {
    vector<vi> g(n);
    for (auto [x, y]: es) {
        g[x].push_back(y);
        g[y].push_back(x);
    }
    vi match(n, -1);

    auto aug = [&](int st) {
        vi fa(n, clr(n, -1), pre(n, -1), tag(n);
        iota(all(fa), 0);
        int tot = 0;
        vi que{st};
        clr[st] = 0;

        function<int(int)> getfa = [&](int x) {
            return fa[x] == x ? x : fa[x] = getfa(fa[x]);
        };

        auto lca = [&](int x, int y) {
            tot++;
            x = getfa(x);
            y = getfa(y);
            while (1) {
                if (x != -1) {
                    if (tag[x] == tot) return x;
                    tag[x] = tot;
                    if (match[x] != -1) x = getfa(pre[match[x]]);
                    else x = -1;
                }
            }
        };
    };
};
```

```
    }
    swap(x, y);
}

};

auto shrink = [&](int x, int y, int f) {
    while (getfa(x) != f) {
        pre[x] = y;
        y = match[x];
        if (clr[y] == 1) {
            clr[y] = 0;
            que.push_back(y);
        }
        if (getfa(x) == x) fa[x] = f;
        if (getfa(y) == y) fa[y] = f;
        x = pre[y];
    }
};

rep(ind, 0, sz(que) - 1) {
    int now = que[ind];
    for (auto v: g[now]) {
        if (getfa(now) == getfa(v) || clr[v] == 1) continue
            //;
        if (clr[v] == -1) {
            clr[v] = 1;
            pre[v] = now;
            if (match[v] == -1) {
                while (now != -1) {
                    int last = match[now];
                    match[now] = v;
                    match[v] = now;
                    if (last != -1) {
                        v = last;
                        now = pre[v];
                    } else break;
                }
                return;
            }
            clr[match[v]] = 0;
            que.push_back(match[v]);
        } else if (clr[v] == 0) {
            assert(getfa(now) != getfa(v));
            int l = lca(now, v);
            shrink(now, v, l);
            shrink(v, now, l);
        }
    }
}

rep(i, 0, n - 1) if (match[i] == -1) aug(i);
return match;
} // hash-cpp-all = cf7d426031408a38af90f44df608495e
```

## hungarian.cpp

**Description:** Given a complete bipartite graph  $G = (L \cup R, E)$ , where  $|L| \leq |R|$ , Finds minimum weighted perfect matching of  $L$ . Returns the matching (a vector of pair<int, int>).  $ws[i][j]$  is the weight of the edge from  $i$ -th vertex in  $L$  to  $j$ -th vertex in  $R$ . Not sure how to choose safe  $T$  since I can not give a bound on values in  $lp$  and  $rp$ . Seems safe to always use **long long**.

**Time:**  $\mathcal{O}(|L|^2|R|)$ .

60 lines

```
template<class T = ll, T INF = numeric_limits<T>::max()>
vector<pii> Hungarian(const vector<vector<T>> &ws) {
    int L = sz(ws), R = L == 0 ? 0 : sz(ws[0]);
    vector<T> lp(L), rp(R); // left & right potential
```

```

vi lm(L, -1), rm(R, -1); // left & right match

rep(i, 0, L - 1) lp[i] = *min_element(all(ws[i]));

auto step = [&](int src) {
    vi que{src}, pre(R, -1); // bfs que & back pointers
    vector<T> sa(R, INF); // slack array; min slack from
    ↪ node in que

    auto extend = [&](int j) {
        if (sa[j] == 0) {
            if (rm[j] == -1) {
                while (j != -1) { // Augment the path
                    int i = pre[j];
                    rm[j] = i;
                    swap(lm[i], j);
                }
                return 1;
            } else que.push_back(rm[j]);
        }
        return 0;
    };

    rep(ind, 0, L - 1) { // BFS to new nodes
        int i = que[ind];
        rep(j, 0, R - 1) {
            if (j == lm[i]) continue;
            T off = ws[i][j] - lp[i] - rp[j]; // Slack in edge
            if (sa[j] > off) {
                sa[j] = off;
                pre[j] = i;
                if (extend(j)) return;
            }
        }
        if (ind == sz(que) - 1) { // Update potentials
            T d = INF;
            rep(j, 0, R - 1) if (sa[j]) d = min(d, sa[j]);

            bool found = 0;
            for (auto i: que) lp[i] += d;
            rep(j, 0, R - 1) {
                if (sa[j]) {
                    sa[j] -= d;
                    if (!found) found |= extend(j);
                } else rp[j] -= d;
            }
            if (found) return;
        }
    };

    rep(i, 0, L - 1) step(i);

    vector<pii> res;
    rep(i, 0, L - 1) res.emplace_back(i, lm[i]);
    return res;
} // hash-cpp-all = ec3fae2f44c4d2e8916ad89e33028e9a

```

## 4.3 Trees

### binary-lifting.cpp

**Description:** Compute the sparse table for binary lifting of a rooted tree  $T$ . The root is set as 0 by default.  $g$  should be the adjacent list of the tree  $T$ .

**Time:**  $\mathcal{O}(|V| \log |V|)$  for precalculation and  $\mathcal{O}(\log |V|)$  for each lca query.

38 lines

```
struct BinaryLifting {
```

```

int n;
vi dep;
vector<vi> anc;

BinaryLifting(const vector<vi> &g, int rt = 0): n(sz(g)),
    ↪ dep(n, -1) {
    assert(n > 0);
    anc.assign(n, vi(__lg(n) + 1));
    auto dfs = [&](auto &dfs, int now, int fa) -> void {
        assert(dep[now] == -1); // make sure it is indeed a
        ↪ tree.
        dep[now] = fa == -1 ? 0 : dep[fa] + 1;
        anc[now][0] = fa;
        rep(i, 1, __lg(n)) {
            anc[now][i] = anc[now][i - 1] == -1 ? -1 : anc[anc[
            ↪ now][i - 1]][i - 1];
        }
        for (auto v: g[now]) if (v != fa) dfs(dfs, v, now);
    };
    dfs(dfs, rt, -1);
}

int swim(int x, int h) {
    for (int i = 0; h && x != -1; h >>= 1, i++) {
        if (h & 1) x = anc[x][i];
    }
    return x;
}

int lca(int x, int y) {
    if (dep[x] < dep[y]) swap(x, y);
    x = swim(x, dep[x] - dep[y]);
    if (x == y) return x;
    for (int i = __lg(n); i >= 0; --i) {
        if (anc[x][i] != anc[y][i]) {
            x = anc[x][i];
            y = anc[y][i];
        }
    }
    return anc[x][0];
} // hash-cpp-all = 49762913e2109a46eal6423cd892c42b

```

### heavy-light-decomposition.cpp

**Description:** Heavy Light Decomposition for a rooted tree  $T$ . The root is set as 0 by default. It can be modified easily for forest.  $g$  should be the adjacent list of the tree  $T$ .  $chainApply(u, v, func, val)$  and  $chainAsk(u, v, func)$  are used for apply / query on the simple path from  $u$  to  $v$  on tree  $T$ .  $func$  is the function you want to use to apply / query on a interval. (Say rangeApply / rangeAsk of Segment tree.)

**Time:**  $\mathcal{O}(|T|)$  for building.  $\mathcal{O}(\log |T|)$  for lca.  $\mathcal{O}(\log |T| \cdot A)$  for chainApply / chainAsk, where  $A$  is the running time of  $func$  in chainApply / chainAsk.

69 lines

```

struct HLD {
    int n; // hash-cpp-1
    vi fa, hson, dfn, dep, top;
    HLD(vvi &g, int rt = 0): n(sz(g)), fa(n, -1), hson(n, -1)
    ↪ , dfn(n), dep(n, 0), top(n) {
        vi siz(n);
        auto dfs = [&](auto &dfs, int now) -> void {
            siz[now] = 1;
            int mx = 0;
            for (auto v: g[now]) if (v != fa[now]) {
                dep[v] = dep[now] + 1;
                fa[v] = now;
                dfs(dfs, v);
                siz[now] += siz[v];
                if (mx < siz[v]) {

```

```

                mx = siz[v];
                hson[now] = v;
            }
        };
        dfs(dfs, rt);

        int cnt = 0;
        auto getdfn = [&](auto &dfs, int now, int sp) {
            top[now] = sp;
            dfn[now] = cnt++;
            if (hson[now] == -1) return;
            dfs(dfs, hson[now], sp);
            for (auto v: g[now]) {
                if (v != hson[now] && v != fa[now]) dfs(dfs, v, v);
            }
        };
        getdfn(getdfn, rt, rt);
    } // hash-cpp-1 = 2568871424fd3facea52f4677941cb68

    int lca(int u, int v) { // hash-cpp-2
        while (top[u] != top[v]) {
            if (dep[top[u]] < dep[top[v]]) swap(u, v);
            u = fa[top[u]];
        }
        if (dep[u] < dep[v]) return u;
        else return v;
    } // hash-cpp-2 = c5c13283ffc68dacc37d3312019a26f8

    template<class... T> // hash-cpp-3
    void chainApply(int u, int v, const function<void(int,
    ↪ int, T...)> &func, const T&... val) {
        int f1 = top[u], f2 = top[v];
        while (f1 != f2) {
            if (dep[f1] < dep[f2]) swap(f1, f2), swap(u, v);
            func(dfn[f1], dfn[u], val...);
            u = fa[f1]; f1 = top[u];
        }
        if (dep[u] < dep[v]) swap(u, v);
        func(dfn[v], dfn[u], val...); // change here if you
        ↪ want the info on edges.
    } // hash-cpp-3 = e995d6fbf54395b102f90775b9a66a89

    template<class T> // hash-cpp-4
    T chainAsk(int u, int v, const function<T(int, int)> &
    ↪ func) {
        int f1 = top[u], f2 = top[v];
        T ans{};
        while (f1 != f2) {
            if (dep[f1] < dep[f2]) swap(f1, f2), swap(u, v);
            ans = ans + func(dfn[f1], dfn[u]);
            u = fa[f1]; f1 = top[u];
        }
        if (dep[u] < dep[v]) swap(u, v);
        ans = ans + func(dfn[v], dfn[u]); // change here if you
        ↪ want the info on edges.
        return ans;
    } // hash-cpp-4 = 65ec12b740accde49b1ac20b95ea1de8
};

```

### centroid-decomposition.cpp

**Description:** Centroid Decomposition of tree  $T$ . Here,  $anc[i]$  is the list of ancestors of vertex  $i$  and the distances to the corresponding ancestor in centroid tree, including itself. Note that the distances are not monotone. Note that the top centroid is in the front of the vector.

**Time:**  $\mathcal{O}(|T| \log |T|)$ .

37 lines



```

struct CentroidDecomposition {
    int n;
    vector<vector<pii>> ancs;

    CentroidDecomposition(vector<vi> &g): n(sz(g)), ancs(n) {
        vi siz(n);
        vector<bool> vis(n);
        auto solve = [&](auto &solve, int st, int tot) -> void
            ↪{
            int mn = 0x3f3f3f3f, cent = -1;
            auto getcent = [&](auto &dfs, int now, int fa) ->
                ↪void {
                siz[now] = 1;
                int mx = 0;
                for (auto v: g[now]) if (v != fa && vis[v] == 0) {
                    dfs(dfs, v, now);
                    siz[now] += siz[v];
                    mx = max(mx, siz[v]);
                }
                mx = max(mx, tot - siz[now]);
                if (mn > mx) mn = mx, cent = now;
            };
            getcent(getcent, st, -1);
            vis[cent] = 1;

            auto dfs = [&](auto &dfs, int now, int fa, int dep)
                ↪-> void {
                ancs[now].emplace_back(cent, dep);
                for (auto v: g[now]) if (v != fa && vis[v] == 0) {
                    dfs(dfs, v, now, dep + 1);
                }
            };
            dfs(dfs, cent, -1, 0);
            // start your work here or inside the function dfs.

            for (auto v: g[cent]) if (vis[v] == 0) solve(solve, v
                ↪, siz[v] < siz[cent] ? siz[v] : tot - siz[cent])
                ↪;
            solve(solve, 0, n);
        };
    }; // hash-cpp-all = 8db9846c598845aeaba8d192e971b266

```

## 4.4 Connectivity

### dsu.cpp

**Description:** Disjoint set union. *merge(x,y)* merges components which *x* and *y* are in respectively and returns 1 if *x* and *y* are in different components.

**Time:** amortized  $\mathcal{O}(\alpha(M,N))$  where *M* is the number of operations. Almost constant in competitive programming.

```

struct DSU {
    vi fa, siz;

    DSU(int n): fa(n), siz(n, 1) { iota(all(fa), 0); }

    int getcomp(int x) {
        return fa[x] == x ? x : fa[x] = getcomp(fa[x]);
    }

    bool merge(int x, int y) {
        int fx = getcomp(x), fy = getcomp(y);
        if (fx == fy) return 0;
        if (siz[fx] < siz[fy]) swap(fx, fy);
        fa[fy] = fx;
        siz[fx] += siz[fy];
        return 1;
    }

```

```

    }
}; // hash-cpp-all = d79908e5926d7bd63f242158624be7d7

```

### undo-dsu.cpp

**Description:** Undoable Disjoint Union Set for set  $0, \dots, N-1$ . Fill in struct *T*, function *join* as well as choosing proper type *Z* for *glob* and remember to initialize it. Use *top = top()* to get a save point; use *undo(top)* to go back to the save point.

**Usage:** UndoDSU dsu(n);

```

...
int top = dsu.top(); // get a save point.
... // do merging and other calculating here.
dsu.undo(top); // get back to the save point.

```

**Time:** Amortized  $\mathcal{O}(\log N)$ .

55 lines

```

struct UndoDSU {
    using Z = int; // choose some proper type (Z) for global
    ↪variable glob.

    struct T {
        int siz;
        // add things you want to maintain here.
        T(int ind = 0): siz(1) {
            // initialize what you add here.
        }
    };

    Z glob;
private:
    void join(T &a, const T& b) {
        a.siz += b.siz;
        // maintain the things you added to struct T.
        // also remember to maintain glob here.
    }

    vi fa;
    vector<T> ts;
    vector<tuple<int, int, T, Z>> sta;
public:
    UndoDSU(int n): fa(n), ts(n) {
        iota(all(fa), 0);
        iota(all(ts), 0);
        // remember initializing glob here.
    }

    int getcomp(int x) {
        while (x != fa[x]) x = fa[x];
        return x;
    }

    bool merge(int x, int y) {
        int fx = getcomp(x), fy = getcomp(y);
        if (fx == fy) return 0;
        if (ts[fx].siz < ts[fy].siz) swap(fx, fy);
        sta.emplace_back(fx, fy, ts[fx], glob);
        fa[fy] = fx;
        join(ts[fx], ts[fy]);
        return 1;
    }

    int top() { return sz(sta); }

    void undo(int top) {
        while (sz(sta) > top) {
            auto &[x, y, dat, g] = sta.back();
            fa[y] = y;
            ts[x] = dat;
            glob = g;
        }
    }

```

```

        sta.pop_back();
    }
}; // hash-cpp-all = 20804d360ba467cdf1cd0b6125550c0f

```

### cut-and-bridge.cpp

**Description:** Given an undirected graph  $G = (V, E)$ , compute all cut vertices and bridges. Cut vertices and bridges are returned in vectors containing indices.

**Time:**  $\mathcal{O}(|V| + |E|)$ .

31 lines

```

auto CutAndBridge(int n, const vector<pii> es) {
    vvi g(n);
    rep(i, 0, sz(es) - 1) {
        auto [x, y] = es[i];
        g[x].push_back(i);
        g[y].push_back(i);
    }

    vi cut, bridge, dfn(n, -1), low(n), mark(sz(es));
    int cnt = 0;
    auto dfs = [&](auto &dfs, int now, int fa) -> void {
        dfn[now] = low[now] = cnt++;
        int sons = 0, isCut = 0;
        for (auto ind: g[now]) if (mark[ind] == 0) {
            mark[ind] = 1;
            auto [x, y] = es[ind];
            int v = now ^ x ^ y;
            if (dfn[v] == -1) {
                sons++;
                dfs(dfs, v, now);
                low[now] = min(low[now], low[v]);
                if (low[v] == dfn[v]) bridge.push_back(ind);
                if (low[v] >= dfn[now] && fa != -1) isCut = 1;
            } else low[now] = min(low[now], dfn[v]);
        }
        if (fa == -1 && sons > 1) isCut = 1;
        if (isCut) cut.push_back(now);
    };
    rep(i, 0, n - 1) if (dfn[i] == -1) dfs(dfs, i, -1);
    return make_tuple(cut, bridge);
}; // hash-cpp-all = c7b8c42c12ad0e48babb6cbda98c1c45

```

### vertex-bcc.cpp

**Description:** Compute the Vertex-BiConnected Components of a graph  $G = (V, E)$  (not necessarily connected). Multiple edges and self loops are allowed. *id[i]* records the index of bcc the *i*-th edge is in. *top[u]* records the second highest vertex (which is unique) in the bcc which vertex *u* is in. (Just for checking if two vertices are in the same bcc.) This code also builds the block forest: *bf* records the edges in the block forest, where the *i*-th bcc corresponds to the  $(n+i)$ -th node. Call *getBlockForest()* to get the adjacency list.

**Time:**  $\mathcal{O}(|V| + |E|)$ .

67 lines

```

struct VertexBCC {
    int n, bcc; // hash-cpp-1
    vi id, top, fa;
    vector<pii> bf; // edges of the block-forest.

    VertexBCC(int n, const vector<pii> &es): n(n), bcc(0), id
        ↪(sz(es)), top(n), fa(n, -1) {
        vvi g(n);
        rep(ind, 0, sz(es) - 1) {
            auto [x, y] = es[ind];
            g[x].push_back(ind);
            g[y].push_back(ind);
        }
    }

```

```

int cnt = 0;
vi dfn(n, -1), low(n), mark(sz(es)), vsta, esta;
auto dfs = [&](auto dfs, int now) -> void {
    low[now] = dfn[now] = cnt++;
    vsta.push_back(now);
    for (auto ind: g[now]) if (mark[ind] == 0) {
        mark[ind] = 1;
        esta.push_back(ind);
        auto [x, y] = es[ind];
        int v = now ^ x ^ y;
        if (dfn[v] == -1) {
            dfs(dfs, v);
            fa[v] = now;
            low[now] = min(low[now], low[v]);
            if (low[v] >= dfn[now]) {
                bf.emplace_back(n + bcc, now);
                while (1) {
                    int z = vsta.back();
                    vsta.pop_back();
                    top[z] = v;
                    bf.emplace_back(n + bcc, z);
                    if (z == v) break;
                }
                while (1) {
                    int z = esta.back();
                    esta.pop_back();
                    id[z] = bcc;
                    if (z == ind) break;
                }
                bcc++;
            }
        } else low[now] = min(low[now], dfn[v]);
    }
};
rep(i, 0, n - 1) if (dfn[i] == -1) {
    dfs(dfs, i);
    top[i] = i;
} // hash-cpp-1 = f2d47f9dcf3538feb29552eef46872dd

bool SameBcc(int x, int y) { // hash-cpp-2
    if (x == fa[top[y]] || y == fa[top[x]]) return 1;
    else return top[x] == top[y];
} // hash-cpp-2 = 3cb78bd6aa7d389b1f6bb850cb631bb2

vector<vi> getBlockForest() { // hash-cpp-3
    vvi g(n + bcc);
    for (auto [x, y]: bf) {
        g[x].push_back(y);
        g[y].push_back(x);
    }
    return g;
} // hash-cpp-3 = 574d110c1d0c530229e4f1b0ee9069d7
};

```

### edge-bcc.cpp

**Description:** Compute the Edge-BiConnected Components of a **connected** graph. Multiple edges and self loops are allowed. Return the size of BCCs and the index of the component each vertex belongs to.  
**Time:**  $\mathcal{O}(|E|)$ .

35 lines

```

auto EdgeBCC(int n, const vector<pii> &es, int st = 0) {
    vi dfn(n, -1), low(n), id(n), mark(sz(es), 0), sta;
    int cnt = 0, bcc = 0;
    vvi g(n);
    rep(ind, 0, sz(es) - 1) {

```

```

        auto [x, y] = es[ind];
        g[x].push_back(ind);
        g[y].push_back(ind);
    }

    auto dfs = [&](auto dfs, int now) -> void {
        low[now] = dfn[now] = cnt++;
        sta.push_back(now);
        for (auto ind: g[now]) if (mark[ind] == 0) {
            mark[ind] = 1;
            auto [x, y] = es[ind];
            int v = now ^ x ^ y;
            if (dfn[v] == -1) {
                dfs(dfs, v);
                low[now] = min(low[now], low[v]);
            } else low[now] = min(low[now], dfn[v]);
        }
        if (low[now] == dfn[now]) {
            while (sta.back() != now) {
                id[sta.back()] = bcc;
                sta.pop_back();
            }
            id[now] = bcc;
            sta.pop_back();
            bcc++;
        }
    };
    dfs(dfs, st);
    return make_tuple(bcc, id);
} // hash-cpp-all = ea66ad6c614370a1b88363aa23f553cd

```

### tarjan.cpp

**Description:** Tarjan algorithm for directed graph  $G = (V, E)$ .

27 lines

```

auto tarjan(const vector<vi> &g) {
    int n = sz(g);
    vi id(n, -1), dfn(n, -1), low(n, -1), sta;
    int cnt = 0, scc = 0;

    auto dfs = [&](auto &dfs, int now) -> void {
        dfn[now] = low[now] = cnt++;
        sta.push_back(now);
        for (auto v: g[now]) {
            if (dfn[v] == -1) {
                dfs(dfs, v);
                low[now] = min(low[now], low[v]);
            } else if (id[v] == -1) low[now] = min(low[now], dfn[v]);
        }
        if (low[now] == dfn[now]) {
            while (1) {
                int z = sta.back();
                sta.pop_back();
                id[z] = scc;
                if (z == now) break;
            }
            scc++;
        }
    };
    rep(i, 0, n - 1) if (dfn[i] == -1) dfs(dfs, i);
    return make_tuple(scc, id);
} // hash-cpp-all = e9681d2c3fd78713716890417a465211

```

### 2sat.cpp

**Description:** 2SAT solver, returns if a 2SAT system of  $V$  variables and  $C$  constraints is satisfiable. If yes, it also gives an assignment. Call `addClause` to add clauses. For example, if you want to add clause  $\neg x \vee y$ , just call `addClause(x, 0, y, 1)`.

**Time:**  $\mathcal{O}(|V| + |C|)$ .

46 lines

```

struct TwoSat {
    int n;
    vector<vi> e;
    vi ans;

    TwoSat(int n): n(n), e(n * 2), ans(n) {}

    void addClause(int x, bool f, int y, bool g) {
        e[x * 2 + !f].push_back(y * 2 + g);
        e[y * 2 + !g].push_back(x * 2 + f);
    }

    bool satisfiable() {
        vi id(n * 2, -1), dfn(n * 2, -1), low(n * 2, -1), sta;
        int cnt = 0, scc = 0;

        auto dfs = [&](auto &dfs, int now) -> void {
            dfn[now] = low[now] = cnt++;
            sta.push_back(now);
            for (auto v: e[now]) {
                if (dfn[v] == -1) {
                    dfs(dfs, v);
                    low[now] = min(low[now], low[v]);
                } else if (id[v] == -1) low[now] = min(low[now],
                    dfn[v]);
            }
            if (low[now] == dfn[now]) {
                while (sta.back() != now) {
                    id[sta.back()] = scc;
                    sta.pop_back();
                }
                id[sta.back()] = scc;
                sta.pop_back();
                scc++;
            }
        };

        rep(i, 0, n * 2 - 1) if (dfn[i] == -1) dfs(dfs, i);
        rep(i, 0, n - 1) {
            if (id[i * 2] == id[i * 2 + 1]) return 0;
            ans[i] = id[i * 2] > id[i * 2 + 1];
        }
        return 1;
    }

    vi getAss() { return ans; }
}; // hash-cpp-all = 48021fb8f8e959774f7a861f2f294deb

```

### link-cut.cpp

1 lines

// TODO

## 4.5 Paths

euler-tour-nonrec.cpp

**Description:** For an edge set  $E$  such that each vertex has an even degree, compute Euler tour for each connected component.  $dir$  indicates edges are directed or not (undirected by default). For undirected graph,  $ori[i]$  records the orientation of the  $i$ -th edge  $es[i] = (x, y)$ , where  $ori[i] = 1$  means  $x \rightarrow y$  and  $ori[i] = -1$  means  $y \rightarrow x$ . Note that this is a non-recursive implementation, which avoids stack size issue on some OJ and also saves memory (roughly saves 2/3 of memory) due to that.

**Time:**  $\mathcal{O}(|V| + |E|)$ .

52 lines

```
struct EulerTour {
    int n;
    vector<vi> tours;
    vi ori;

    EulerTour(int n, const vector<pii> &es, int dir = 0): n(n) {
        ori(sz(es)) {
            vector<vi> g(n);
            int m = sz(es);
            rep(i, 0, m - 1) {
                auto [x, y] = es[i];
                g[x].push_back(i);
                if (dir == 0) g[y].push_back(i);
            }

            vi path, cur(n);
            vector<pii> sta;
            auto solve = [&](int st) {
                sta.emplace_back(st, -1);
                while (sz(sta)) {
                    auto [now, pre] = sta.back();
                    int fin = 1;
                    for (int &i = cur[now]; i < sz(g[now]); ) {
                        auto ind = g[now][i++];
                        if (ori[ind]) continue;
                        auto [x, y] = es[ind];
                        ori[ind] = x == now ? 1 : -1;
                        int v = now ^ x ^ y;
                        sta.emplace_back(v, ind);
                        fin = 0;
                        break;
                    }
                }
                if (fin) {
                    if (pre != -1) path.push_back(pre);
                    sta.pop_back();
                }
            };

            rep(i, 0, n - 1) {
                path.clear();
                solve(i);
                if (sz(path)) {
                    reverse(all(path));
                    tours.push_back(path);
                }
            }

            vector<vi> getTours() { return tours; }

            vi getOrient() { return ori; }
}; // hash-cpp-all = e5f7e9e86d4e1d9d5aa0be753a0cb6e9
```

## 5.1 String Matching

### kmp.cpp

**Description:** Compute fail table of pattern string  $s = s_0 \dots s_{n-1}$  in linear time and get all matched positions in text string  $t$  in linear time.  $fail[i]$  denotes the length of the border of substring  $s_0 \dots s_i$ . In  $match(t)$ ,  $res[i] = 1$  means that  $t_i \dots t_{i+n-1}$  matched to  $s$ .

**Usage:** KMP kmp(s); //  $s$  can be string or vector.

**Time:**  $\mathcal{O}(|s|)$  for precalculation and  $\mathcal{O}(|t|)$  for matching.

25 lines

```
template<class T> struct KMP {
    const T s; // hash-cpp-1
    int n;
    vi fail;

    KMP(const T &s): s(s), n(sz(s)), fail(n) {
        int j = 0;
        rep(i, 1, n - 1) {
            while (j > 0 && s[j] != s[i]) j = fail[j - 1];
            if (s[j] == s[i]) j++;
            fail[i] = j;
        }
    } // hash-cpp-1 = abad2ebf1bb7e6689c795bf074babcc6

    vi match(const T &t) { // hash-cpp-2
        int m = sz(t), j = 0;
        vi res(m);
        rep(i, 0, m - 1) {
            while (j > 0 && (j == n || s[j] != t[i])) j = fail[j - 1];
            if (s[j] == t[i]) j++;
            if (j == n) res[i - n + 1] = 1;
        }
        return res;
    } // hash-cpp-2 = f586c1dee3650d26ab1db15140981c8b
};
```

### z-algo.cpp

**Description:** Given string  $s = s_0 \dots s_{n-1}$ , compute array  $z$  where  $z[i]$  is the lcp of  $s_0 \dots s_{n-1}$  and  $s_i \dots s_{n-1}$ . Use function  $cal(t)$  (where  $|t| = m$ ) to calculate the lcp of  $s_0 \dots s_{n-1}$  and  $t_i \dots t_{m-1}$  for each  $i$ .

**Usage:** ZAlgo za(s); //  $s$  can be string or vector.

**Time:**  $\mathcal{O}(|s|)$  for precalculation and  $\mathcal{O}(|t|)$  for matching.

34 lines

```
template<class T>
struct ZAlgo {
    const T s; // hash-cpp-1
    int n;
    vi z;

    ZAlgo(const T &s): s(s), n(sz(s)), z(n) {
        z[0] = n;
        int l = 0, r = 0;
        rep(i, 1, n - 1) {
            z[i] = max(0, min(z[i - 1], r - i));
            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];
            }
        }
    } // hash-cpp-1 = 0a5f9be882b336b6aa27f9ee79d633ec

    vi cal(const T &t) { // hash-cpp-2
        int m = sz(t);
        vi res(m);
        int l = 0, r = 0;

```

```
rep(i, 0, m - 1) {
    res[i] = max(0, min(i - 1 < n ? z[i - 1] : 0, r - i))
    while (i + res[i] < m && s[res[i]] == t[i + res[i]])
        res[i]++;
    if (i + res[i] > r) {
        l = i;
        r = i + res[i];
    }
}
return res;
} // hash-cpp-2 = 0a29c792be96f8c1ccdb699df9cfc984
};
```

### aho-corasick.cpp

**Description:** Aho Corasick Automaton of strings  $s_0, \dots, s_{n-1}$ . Call  $build()$  after you insert all strings  $s_0, \dots, s_{n-1}$ .

**Usage:** AhoCorasick<'a', 26> ac; // for strings consisting of lowercase letters.

ac.insert("abc"); // insert string "abc".

ac.insert("acc"); // insert string "acc".

ac.build();

**Time:**  $\mathcal{O}(\sum_{i=0}^{n-1} |s_i|)$ .

48 lines

```
template<char st, int C>
struct AhoCorasick {
    struct node {
        int nxt[C];
        int fail;
        int cnt;
        node() {
            memset(nxt, -1, sizeof nxt);
            fail = -1;
            cnt = 0;
        }
    };

    vector<node> t;

    AhoCorasick(): t(1) {}

    int insert(const string &s) {
        int now = 0;
        for (auto ch: s) {
            int c = ch - st;
            if (t[now].nxt[c] == -1) {
                t.emplace_back();
                t[now].nxt[c] = sz(t) - 1;
            }
            now = t[now].nxt[c];
        }
        t[now].cnt++;
        return now;
    }

    void build() {
        vi que(0);
        rep(ind, 0, sz(que) - 1) {
            int now = que[ind], fa = t[now].fail;
            rep(c, 0, C - 1) {
                int &v = t[now].nxt[c];
                int u = fa == -1 ? 0 : t[fa].nxt[c];
                if (v == -1) v = u;
                else {
                    t[v].fail = u;
                    que.push_back(v);
                }
            }
        }
    }
};
```

```

    }
    if (fa != -1) t[now].cnt += t[fa].cnt;
}
}; // hash-cpp-all = 3dca34c2bb5ab364d7abcab29a8c27f4

```

## 5.2 Suffices & Substrings

### suffix-array.cpp

**Description:** Suffix Array for non-cyclic string  $s = s_0 \dots s_{n-1}$ .  $rank[i]$  records the rank of the  $i$ -th suffix  $s_i \dots s_{n-1}$ .  $sa[i]$  records the starting position of the  $i$ -th smallest suffix.  $h[i]$  (also called height array or lcp array) records the lcp of the  $sa[i]$ -th suffix and the  $sa[i+1]$ -th suffix in  $s$ .

**Usage:** SA suf(s); //  $s$  can be string or vector.

**Time:**  $\mathcal{O}(|s| \log |s|)$ .

49 lines

```

struct SA {
    int n;
    vi str, sa, rank, h;

    template<class T> SA(const T &s): n(sz(s)), str(n + 1),
        ↪ sa(n + 1), rank(n + 1), h(n - 1) {
        auto vec = s;
        sort(all(vec)); vec.erase(unique(all(vec)), vec.end());
        rep(i, 0, n - 1) str[i] = rank[i] = lower_bound(all(vec)
            ↪, s[i]) - vec.begin() + 1;
        iota(all(sa), 0);
        n++;

        for (int len = 0; len < n; len = len ? len * 2 : 1) {
            vi cnt(n + 1);
            for (auto v : rank) cnt[v + 1]++;
            rep(i, 1, n - 1) cnt[i] += cnt[i - 1];

            vi nsa(n), nrank(n);

            for (auto pos: sa) {
                pos -= len;
                if (pos < 0) pos += n;
                nsa[cnt[rank[pos]]++] = pos;
            }
            swap(sa, nsa);

            int r = 0, oldp = -1;
            for (auto p: sa) {
                auto next = [&](int a, int b) { return a + b < n ?
                    ↪ a + b : a + b - n; };
                if (~oldp) r += rank[p] != rank[oldp] || rank[next(
                    ↪ p, len)] != rank[next(oldp, len)];
                nrank[p] = r;
                oldp = p;
            }
            swap(rank, nrank);
        }
        sa = vi(sa.begin() + 1, sa.end());
        rank.resize(--n);
        rep(i, 0, n - 1) rank[sa[i]] = i;

        // compute height array.
        int len = 0;
        rep(i, 0, n - 1) {
            if (len) len--;
            int rk = rank[i];
            if (rk == n - 1) continue;
            while (str[i + len] == str[sa[rk + 1] + len]) len++;
            h[rk] = len;
        }
    }
}

```

```

    }
}; // hash-cpp-all = dc03be590b13b29f57b3250dc4634be7

```

### suffix-array-lcp.cpp

**Description:** Suffix Array with sparse table answering lcp of suffices.

**Usage:** SA suf(s); //  $s$  can be string or vector.

**Time:**  $\mathcal{O}(|s| \log |s|)$  for construction.  $\mathcal{O}(1)$  per query.

"suffix-array.cpp" 22 lines

```

struct SA_lcp: SA {
    vector<vi> st;

    template<class T> SA_lcp(const T &s): SA(s) {
        assert(n > 0);
        st.assign(__lg(n) + 1, vi(n));
        st[0] = h;
        st[0].push_back(0); // just to make st[0] of size n.
        rep(i, 1, __lg(n)) rep(j, 0, n - (1 << i)) {
            st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i -
                ↪ 1))]);
        }
        // return lcp(suff_i, suff_j) for i != j.
        int lcp(int i, int j) {
            if (i == n || j == n) return 0;
            assert(i != j);
            int l = rank[i], r = rank[j];
            if (l > r) swap(l, r);
            int k = __lg(r - l);
            return min(st[k][l], st[k][r - (1 << k)]);
        }
    }; // hash-cpp-all = ff57ad558a18576768e4c3b01e315c93
}

```

### sam.cpp

**Description:** Suffix Automaton of a given string  $s$ . (Using map to store sons makes it 2~3 times slower but it should be fine in most cases.)  $len$  is the length of the longest substring corresponding to the state.  $fa$  is the father in the prefix tree. Note that  $fa[i] < i$  doesn't hold.  $occ$  is 0/1, indicating if the state contains a prefix of the string  $s$ . One can do a dfs/bfs to compute for each substring, how many times it occurs in the whole string  $s$ . (See function `calOccurrence` for bfs implementation.) root is set as 0.

**Usage:** SAM sam(s); //  $s$  can be string or vector<int>.

**Time:**  $\mathcal{O}(|s|)$ .

74 lines

```

template<class T> struct SAM {
    struct node { // hash-cpp-1
        map<int, int> nxt; // change this if it is slow.
        int fa, len;
        int occ, pos; // # of occurrence (as prefix) & endpos.
        node(int fa = -1, int len = 0): fa(fa), len(len) {
            occ = pos = 0;
        }
    };

    T s;
    int n;
    vector<node> t;
    vi at; // at[i] = the state at which the i-th prefix of s
        ↪ is.

    SAM(const T &s): s(s), n(sz(s)), at(n) {
        t.emplace_back();
        int last = 0; // create root.

        auto ins = [&](int i, int c) {
            int now = last;
            t.emplace_back(-1, t[now].len + 1);

```

```

last = sz(t) - 1;
t[last].occ = 1;
t[last].pos = i;
at[i] = last;

```

```

while (now != -1 && t[now].nxt.count(c) == 0) {
    t[now].nxt[c] = last;
    now = t[now].fa;
}
if (now == -1) t[last].fa = 0; // root is 0.
else {
    int p = t[now].nxt[c];
    if (t[p].len == t[now].len + 1) t[last].fa = p;
    else {
        auto tmp = t[p];
        tmp.len = t[now].len + 1;
        tmp.occ = 0; // do not copy occ.
        t.push_back(tmp);
        int np = sz(t) - 1;

        t[last].fa = t[p].fa = np;
        while (now != -1 && t[now].nxt.count(c) && t[now]
            ↪ .nxt[c] == p) {
            t[now].nxt[c] = np;
            now = t[now].fa;
        }
    }
};

```

```

rep(i, 0, n - 1) ins(i, s[i]);
} // hash-cpp-1 = 1c12eb7fbeecc418a5befc77214c19b9b

```

```

void calOccurrence() { // hash-cpp-2
    vi sum(n + 1), que(sz(t));
    for (auto &it: t) sum[it.len]++;
    rep(i, 1, n) sum[i] += sum[i - 1];
    rep(i, 0, sz(t) - 1) que[--sum[t[i].len]] = i;
    reverse(all(que));
    for (auto now: que) if (now != 0) t[t[now].fa].occ += t
        ↪ [now].occ;
} // hash-cpp-2 = 34e98c4d6ea1e86aa5d52a582becf8a8

```

```

vector<vi> ReversedPrefixTree() { // hash-cpp-3
    vector<vi> g(sz(t));
    rep(now, 1, sz(t) - 1) g[t[now].fa].push_back(now);
    rep(now, 0, sz(t) - 1) {
        sort(all(g[now]), [&](int i, int j) {
            return s[t[i].pos - t[now].len] < s[t[j].pos - t
                ↪ [now].len];
        });
    }
    return g;
} // hash-cpp-3 = aadc726973415dfaacc1e483d8fac558b
};

```

### general-sam.cpp

**Description:** General Suffix Automaton of a given Trie  $T$ . (Using map to store sons makes it 2~3 times slower but it should be fine in most cases. If  $T$  is of size  $> 10^6$ , then you should think of using `int[]` instead of `map`.)  $len$  is the length of the longest substring corresponding to the state.  $fa$  is the father in the reversed prefix tree. Note that  $fa[i] < i$  doesn't hold.  $occ$  should be set manually when building Trie  $T$ . root is 0.

**Usage:** GSAM sam(T); //  $T$  should be vector<GSAM::node>.

**Time:**  $\mathcal{O}(|T|)$ .

52 lines

```

struct GSAM {
    struct node {
        map<int, int> nxt; // change this if it is slow.
        int fa, len;
        int occ;
        node() { fa = -1; len = occ = 0; }
    };

    vector<node> t;
    GSAM(const vector<node> &trie): t(trie) { // swap(t, trie)
        here if TL and ML is tight
        auto ins = [&](int now, int c) {
            int last = t[now].nxt[c];
            t[last].len = t[now].len + 1;
            now = t[now].fa;
            while (now != -1 && t[now].nxt.count(c) == 0) {
                t[now].nxt[c] = last;
                now = t[now].fa;
            }
            if (now == -1) t[last].fa = 0;
            else {
                int p = t[now].nxt[c];
                if (t[p].len == t[now].len + 1) t[last].fa = p;
                else { // clone a node np from node p.
                    t.emplace_back();
                    int np = sz(t) - 1;
                    for (auto [i, v]: t[p].nxt) if (t[v].len > 0) {
                        t[np].nxt[i] = v; // use emplace here?
                    }
                    t[np].fa = t[p].fa;
                    t[np].len = t[now].len + 1;

                    t[last].fa = t[p].fa = np;
                    while (now != -1 && t[now].nxt.count(c) && t[now]
                        .nxt[c] == p) {
                        t[now].nxt[c] = np;
                        now = t[now].fa;
                    }
                }
            }
        };

        vi que{0};
        rep(ind, 0, sz(que) - 1) {
            int now = que[ind];
            vi cs;
            for (auto [c, v]: t[now].nxt) {
                cs.push_back(c);
                que.push_back(v);
            }
            for (auto c: cs) ins(now, c);
        }
    };

    vi que{0};
    rep(ind, 0, sz(que) - 1) {
        int now = que[ind];
        vi cs;
        for (auto [c, v]: t[now].nxt) {
            cs.push_back(c);
            que.push_back(v);
        }
        for (auto c: cs) ins(now, c);
    }
};

vi que{0};
rep(ind, 0, sz(que) - 1) {
    int now = que[ind];
    vi cs;
    for (auto [c, v]: t[now].nxt) {
        cs.push_back(c);
        que.push_back(v);
    }
    for (auto c: cs) ins(now, c);
}
}; // hash-cpp-all = add4c78221df38584b76536f66703db7

```

### lyndon-factorization.cpp

**Description:** Lyndon factorization of string  $s$ . Return a vector of pairs  $(l, r)$ , representing substring  $s_l \dots s_r$ .

**Time:**  $\mathcal{O}(|s|)$ . 17 lines

```

vector<pii> duval(string const& s) {
    int n = sz(s), i = 0;
    vector<pii> res;
    while (i < n) {
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j]) k = j;

```

```

        else k++;
        j++;
    }
    while (i <= k) {
        res.emplace_back(i, i + j - k - 1);
        i += j - k;
    }
}

return res;
}; // hash-cpp-all = 6fff07a96ae3b4e5c66e847abfeb48c6

```

## 5.3 Palindromes

### manacher.cpp

**Description:** Manacher Algorithm for finding all palindrome substrings of  $s = s_0 \dots s_{n-1}$ .  $s$  can actually be string or vector (say  $\text{vector}<\text{int}>$ ). For returned vector  $len$ ,  $len[i*2] = r$  means that  $s_{i-r+1} \dots s_{i+r-1}$  is the maximal palindrome centered at position  $i$ .  $len[i*2+1] = r$  means that  $s_{i-r+1} \dots s_{i+r}$  is the maximal palindrome centered between position  $i$  and  $i+1$ .

**Usage:**  $\text{vi rs} = \text{Manacher}(s)$ ; //  $s$  can be string or vector.

**Time:**  $\mathcal{O}(|s|)$ . 12 lines

```

template<class T>
vi Manacher(const T &s) {
    int n = sz(s), j = 0;
    vi len(n * 2 - 1, 1);
    rep(i, 1, n * 2 - 2) {
        int p = i / 2, q = i - p, r = (j + 1) / 2 + len[j] - 1;
        len[i] = r < q ? 0 : min(r - q + 1, len[j * 2 - i]);
        while (p > len[i] - 1 && q + len[i] < n && s[p - len[i]
            .nxt[q + len[i]]] == s[q + len[i]]) len[i]++;
        if (q + len[i] - 1 > r) j = i;
    }
    return len;
}; // hash-cpp-all = 4c6da773ee61b4d53dd654a4d0d04a4c

```

### palindrome-tree.cpp

**Description:** Given string  $s = s_0 \dots s_{n-1}$ , build the palindrom tree (automaton) for  $s$ . Each state of the automaton corresponds to a palindrome substring of  $s$ .  $t[i].fail$  is the state which is a border of state  $i$ . Note that  $t[i].fail < i$  holds.

**Usage:**  $\text{Palindrome pt}(s)$ ; //  $s$  can be string or vector.

**Time:**  $\mathcal{O}(|s|)$ . 36 lines

```

struct PalindromeTree {
    struct node {
        map<int, int> nxt;
        int fail, len;
        int cnt;
        node(int fail, int len): fail(fail), len(len) {
            cnt = 0;
        }
    };

    vector<node> t;

    template<class T>
    PalindromeTree(const T &s) {
        int n = sz(s);
        t.emplace_back(-1, -1); // Odd root -> state 0.
        t.emplace_back(0, 0); // Even root -> state 1.

```

```

        int now = 0;
        auto ins = [&](int pos) {
            auto get = [&](int i) {
                while (pos == t[i].len || s[pos - 1 - t[i].len] !=
                    s[pos]) i = t[i].fail;
                return i;
            };

```

```

        };
        int c = s[pos];
        now = get(now);
        if (t[now].nxt.count(c) == 0) {
            int q = now == 0 ? 1 : t[get(t[now].fail)].nxt[c];
            t.emplace_back(q, t[now].len + 2);
            t[now].nxt[c] = sz(t) - 1;
        }
        now = t[now].nxt[c];
        t[now].cnt++;
    };
    rep(i, 0, n - 1) ins(i);
}; // hash-cpp-all = ca74a23e6dec05d3f4328aa98fd3d4d3

```

## 5.4 Hashes

### hash-struct.cpp

**Description:** Hash struct. 1000000007 and 1000050131 are good moduli. 19 lines

```

template<int m1, int m2>
struct Hash {
    int x, y;
    Hash(ll a, ll b): x(a % m1), y(b % m2) {
        if (x < 0) x += m1;
        if (y < 0) y += m2;
    }
    Hash(ll a = 0): Hash(a, a) {}

    using H = Hash;
    static int norm(int x, int mod) { return x >= mod ? x -
        mod : x < 0 ? x + mod : x; }
    friend H operator +(H a, H b) { a.x = norm(a.x + b.x, m1);
        a.y = norm(a.y + b.y, m2); return a; }
    friend H operator -(H a, H b) { a.x = norm(a.x - b.x, m1);
        a.y = norm(a.y - b.y, m2); return a; }
    friend H operator *(H a, H b) { return H{lll * a.x * b.x,
        lll * a.y * b.y}; }

    friend bool operator ==(H a, H b) { return tie(a.x, a.y)
        == tie(b.x, b.y); }
    friend bool operator !=(H a, H b) { return tie(a.x, a.y)
        != tie(b.x, b.y); }
    friend bool operator <(H a, H b) { return tie(a.x, a.y) <
        tie(b.x, b.y); }
}; // hash-cpp-all = ff126b1c842614ecc3db2080807d765e

```

### string-hash.cpp

**Description:** Hash of a string.

**Usage:**  $\text{StringHash}<\text{unsigned long long}> \text{ha}(s)$ ; //  $s$  can be string or  $\text{vector}<\text{int}>$ .

**Time:**  $\mathcal{O}(|s|)$ . 15 lines

```

template<class hashv>
struct StringHash {
    const hashv base = 131; // change this if you hash a
        vector<int>.
    int n;
    vector<hashv> hs, pw;

    template<class T>
    StringHash(const T &s): n(sz(s)), hs(n + 1), pw(n + 1) {
        pw[0] = 1;
        rep(i, 1, n) pw[i] = pw[i - 1] * base;
        rep(i, 0, n - 1) hs[i + 1] = hs[i] * base + s[i];
    }

```

```

hashv get(int l, int r) { return hs[r + 1] - hs[l] * pw[r
    ↪ + 1 - 1]; }
}; // hash-cpp-all = 6575c218c608958f097a71917dab22a9

```

## de-bruijin.cpp

1 lines

// TODO

## Numerical (6)

## 6.1 Transforms &amp; Polynomials

## fft.cpp

**Description:** Fast Fourier Transform.  $T$  can be **double** or **long double**.

**Usage:** FFT<double> fft;

```

auto cs = fft.conv(vector<double>{1, 2, 3},
vector<double>{3, 4, 5});
vector<int> ds = fft.conv(vector<int>{1, 2, 3},
vector<int>{3, 4, 5}, 1000000007); // convolution of
integers wrt arbitrary  $mod \leq 2^{31} - 1$ .

```

**Time:**  $\mathcal{O}(N \log N)$ .

73 lines

```

template<class T>
struct FFT {
    using cp = complex<T>;
    static constexpr T pi = acos(T{-1});
    vi r;
    int n2;

    void dft(vector<cp> &a, int is_inv) { // is_inv == 1 ->
        ↪ idft.
        rep(i, 1, n2 - 1) if (r[i] > i) swap(a[i], a[r[i]]);
        for(int step = 1; step < n2; step <= 1) {
            vector<cp> w(step);
            rep(j, 0, step-1) { // this has higher precision,
                ↪ compared to using the power of zeta.
                T theta = pi * j / step;
                if (is_inv) theta = -theta;
                w[j] = cp{cos(theta), sin(theta)};
            }
            for (int i = 0; i < n2; i += step << 1) {
                rep(j, 0, step - 1) {
                    cp tmp = w[j] * a[i + j + step];
                    a[i + j + step] = a[i + j] - tmp;
                    a[i + j] += tmp;
                }
            }
        }
        if (is_inv) {
            for (auto &x: a) x /= n2;
        }
    }

    void pre(int n) { // set n2, r;
        int len = 0;
        for (n2 = 1; n2 < n; n2 <= 1) len++;
        r.resize(n2);
        rep(i, 1, n2 - 1) r[i] = (r[i >> 1] >> 1) | ((i & 1) <<
            ↪ (len - 1));
    }

    template<class Z> vector<Z> conv(const vector<Z> &A,
        ↪ const vector<Z> &B) {
        int n = sz(A) + sz(B) - 1;
        pre(n);
        vector<cp> a(n2, 0), b(n2, 0);
        rep(i, 0, sz(A) - 1) a[i] = A[i];
    }
}

```

```

rep(i, 0, sz(B) - 1) b[i] = B[i];

dft(a, 0); dft(b, 0);
rep(i, 0, n2 - 1) a[i] *= b[i];
dft(a, 1);
vector<Z> res(n);
T eps = T{0.5} * (static_cast<Z>(1e-9) == 0);
rep(i, 0, n - 1) res[i] = a[i].real() + eps;
return res;
}

vi conv(const vi &A, const vi &B, int mod) {
    int M = sqrt(mod) + 0.5;
    int n = sz(A) + sz(B) - 1;
    pre(n);
    vector<cp> a(n2, 0), b(n2, 0), c(n2, 0), d(n2, 0);
    rep(i, 0, sz(A) - 1) a[i] = A[i] / M, b[i] = A[i] % M;
    rep(i, 0, sz(B) - 1) c[i] = B[i] / M, d[i] = B[i] % M;

    dft(a, 0); dft(b, 0); dft(c, 0); dft(d, 0);
    vi res(n);

    auto work = [&](vector<cp> &a, vector<cp> &b, int w,
        ↪ int mod) {
        vector<cp> tmp(n2);
        rep(i, 0, n2 - 1) tmp[i] = a[i] * b[i];
        dft(tmp, 1);
        rep(i, 0, n - 1) res[i] = (res[i] + (ll)tmp[i].real
            ↪ () + 0.5) % mod * w % mod;
    };
    work(a, c, ll * M * M % mod, mod);
    work(b, d, 1, mod);
    work(a, d, M, mod);
    work(b, c, M, mod);
    return res;
}

}; // hash-cpp-all = 9e4b0b0ed2a6597eef170ecd23137484

```

## ntt.cpp

**Description:** Number Theoretic Transform. class  $T$  should have static function  $getMod()$  to provide the  $mod$ . We usually just use  $modnum$  as the template parameter. To keep the code short we just set the primitive root as 3. However, it might be wrong when  $mod \neq 998244353$ . Here are some commonly used  $mods$  and the corresponding primitive root.

$g \rightarrow mod(\max \log(n))$ :  
 $3 \rightarrow 104857601$  (22),  $167772161$  (25),  $469762049$  (26),  $998244353$  (23),  
 $1004535809$  (21);  
 $10 \rightarrow 786433$  (18);  
 $31 \rightarrow 2013265921$  (27).

**Usage:** const int mod = 998244353;  
 using Mint = Z<mod>; // Z is modnum struct.

```

...
FFT<Mint> ntt(3); // use 3 as primitive root.
vector<Mint> as = ntt.conv(vector<Mint>{1, 2, 3},
vector<Mint>{2, 3, 4});

```

**Time:**  $\mathcal{O}(N \log N)$ .

51 lines

```

template<class T>
struct FFT {
    const T g; // primitive root.
    vi r;
    int n2;

    FFT(T _g = 3): g(_g) {}

    void dft(vector<T> &a, int is_inv) { // is_inv == 1 ->
        ↪ idft.
        rep(i, 1, n2 - 1) if (r[i] > i) swap(a[i], a[r[i]]);
    }
}

```

```

for(int step = 1; step < n2; step <= 1) {
    vector<T> w(step);
    T zeta = g.pow((T::getMod() - 1) / (step << 1));
    if (is_inv) zeta = 1 / zeta;
}

```

```

w[0] = 1;
rep(i, 1, step - 1) w[i] = w[i - 1] * zeta;
for (int i = 0; i < n2; i += step << 1) {
    rep(j, 0, step - 1) {
        T tmp = w[j] * a[i + j + step];
        a[i + j + step] = a[i + j] - tmp;
        a[i + j] += tmp;
    }
}
}

```

```

if (is_inv == 1) {
    T inv = T{1} / n2;
    rep(i, 0, n2 - 1) a[i] *= inv;
}
}

```

```

void pre(int n) { // set n2, r; also used in polynomial
    ↪ inverse.
    int len = 0;
    for (n2 = 1; n2 < n; n2 <= 1) len++;
    r.resize(n2);
    rep(i, 1, n2 - 1) r[i] = (r[i >> 1] >> 1) | ((i & 1) <<
        ↪ (len - 1));
}

```

```

vector<T> conv(vector<T> a, vector<T> b) {
    int n = sz(a) + sz(b) - 1;
    pre(n);
    a.resize(n2, 0);
    b.resize(n2, 0);
    dft(a, 0); dft(b, 0);
    rep(i, 0, n2 - 1) a[i] *= b[i];
    dft(a, 1);
    a.resize(n);
    return a;
}

}; // hash-cpp-all = c79d81db99fdb79f856409c48821f21c

```

## polynomial.cpp

**Description:** Basic polynomial struct. Usually we use  $modnum$  as template parameter.  $inv(k)$  gives the inverse of the polynomial  $mod x^k$  (by default  $k$  is the highest power plus one).

48 lines

```

template<class T>
struct poly: vector<T> {
    using vector<T>::vector; // hash-cpp-1
    poly(const vector<T> &vec): vector<T>(vec) {}

    friend poly& operator+=(poly &a, const poly &b) {
        FFT<T> fft;
        a = fft.conv(a, b);
        return a;
    }

    friend poly operator*(const poly &a, const poly &b) {
        ↪ auto c = a; return c *= b; }

    poly inv(int n = 0) const {
        const poly &f = *this;
        assert(sz(f) > 0);
        if (n == 0) n = sz(*this);
        poly res(1 / f[0]);
    }
}

```



```

FFT<T> fft;
for (int m = 2; m < n * 2; m <= 1) {
    poly a(f.begin(), f.begin() + m);
    a.resize(m * 2, 0);
    res.resize(m * 2, 0);
    fft.pre(m * 2);
    fft.dft(a, 0); fft.dft(res, 0);
    rep(i, 0, m * 2 - 1) res[i] = (2 - a[i] * res[i]) *
        ↪ res[i];
    fft.dft(res, 1);
    res.resize(m);
}
res.resize(n);
return res;
} // hash-cpp-1 = 9cecbacfe9d0d397fd8701b6594f8045

// the following is seldom used.
friend poly& operator +=(poly &a, const poly &b) { //
    ↪ hash-cpp-2
    if (sz(a) < sz(b)) a.resize(sz(b), 0);
    rep(i, 0, sz(b) - 1) a[i] += b[i];
    return a;
}
friend poly operator +(const poly &a, const poly &b) {
    ↪ auto c = a; return c += b; }

friend poly& operator -=(poly &a, const poly &b) {
    if (sz(a) < sz(b)) a.resize(sz(b), 0);
    rep(i, 0, sz(b) - 1) a[i] -= b[i];
    return a;
}
friend poly operator -(const poly &a, const poly &b) {
    ↪ auto c = a; return c -= b; }
// hash-cpp-2 = a4c680e717c3d8a211115bef9fb73e1e
};

```

### linear-recurrence-kth-term.cpp

**Description:** Suppose  $a_i = \sum_{j=1}^d c_j * a_{i-j}$ , then just let  $A = \{a_0, \dots, a_{d-1}\}$  and  $C = \{c_1, \dots, c_d\}$ . Here is how it works. Let  $Q(x)$  be the characteristic polynomial of our recurrence, and  $F(x) = \sum_{i=0}^{\infty} a_i x^i$  be the generating formal power series of our sequence. Then it can be seen that all nonzero terms of  $F(x)Q(x)$  are of at most  $(n-1)$ -st power. This means that  $F(x) = P(x)/Q(x)$  for some polynomial  $P(x)$ . Moreover, we know what  $P(x)$  is: it is basically the first  $n$  terms of  $F(x)Q(x)$ , that is, can be found in one multiplication of  $a_0 + \dots + a_{n-1}x^{n-1}$  and  $Q(x)$ , and then trimming to the proper degree.

**Time:**  $\mathcal{O}(d \log^2 d)$ .

"polynomial.cpp" 26 lines

```

template<class T>
T fps_coeff(poly<T> P, poly<T> Q, ll k) {
    while (k >= sz(Q)) {
        auto nQ(Q);
        rep(i, 0, sz(nQ) - 1) if (i & 1) nQ[i] = 0 - nQ[i];
        auto PQ = P * nQ;
        auto Q2 = Q * nQ;
        poly<T> R, S;
        rep(i, 0, sz(PQ) - 1) if ((k + i) % 2 == 0) R.push_back
            ↪ (PQ[i]);
        rep(i, 0, sz(Q2) - 1) if (i % 2 == 0) S.push_back(Q2[i]
            ↪ );
        swap(P, R);
        swap(Q, S);
        k >>= 1;
    }
}

```

```

return (P * Q.inv())[k];
}

template<class T>
T linear_rec_kth(const poly<T> &A, const poly<T> &C, ll k)
    ↪ {
    poly<T> Q{1}; // Q is characteristic polynomial.
    for (auto x: C) Q.push_back(0 - x);
    auto P = A * Q;
    P.resize(sz(Q) - 1);
    return fps_coeff(P, Q, k);
} // hash-cpp-all = 320c2d19b585cfcec2a2bd545b5b8d99

```

### berlekamp-massey.cpp

// TODO

### fast-subset-transform.cpp

**Description:** Fast Subset Transform, which is also known as fast zeta transform. Length of  $a$  should be a power of 2.

**Time:**  $\mathcal{O}(N \log N)$ , where  $N$  is the length of  $a$ .

10 lines

```

template<class T>
void fst(vector<T> &a, int is_inv) {
    int n = sz(a);
    for (int s = 1; s < n; s <= 1) {
        rep(i, 0, n - 1) if (i & s) {
            if (is_inv == 0) a[i] += a[i ^ s];
            else a[i] -= a[i ^ s];
        }
    }
} // hash-cpp-all = 06f39b727394293d6d6f6bbf5ac467db

```

### subset-convolution.cpp

**Description:** Subset Convolution of array  $a$  and  $b$ . Resulting array  $c$  satisfies  $c_z = \sum_{x,y: x|y=z, x \& y=0} a_x \cdot b_y$ . Length of  $a$  and  $b$  should be same and be a power of 2.

**Time:**  $\mathcal{O}(N \log^2 N)$ , where  $N$  is the length of  $a$ .

"fast-subset-transform.cpp" 22 lines

```

template<class T>
vector<T> SubsetConv(const vector<T> &as, const vector<T> &
    ↪ bs) {
    int n = sz(as);
    assert(n > 0 && sz(bs) == n);
    int k = __lg(n);
    vector<vector<T>> ps(k + 1, vector<T>(n)), qs(ps), rs(ps)
        ↪ ;
    rep(x, 0, n - 1) {
        ps[__builtin_popcount(x)][x] = as[x];
        qs[__builtin_popcount(x)][x] = bs[x];
    }
    for (auto &vec: ps) fst(vec, 0);
    for (auto &vec: qs) fst(vec, 0);
    rep(i, 0, k) rep(j, 0, k - i) {
        rep(x, 0, n - 1) rs[i + j][x] += ps[i][x] * qs[j][x];
    }
    for (auto &vec: rs) fst(vec, 1);
    vector<T> cs(n);
    rep(x, 0, n - 1) {
        cs[x] = rs[__builtin_popcount(x)][x];
    }
    return cs;
} // hash-cpp-all = 79c3cbd63fd24f3ecd9f93c66746f2ac

```

### fwht.cpp

**Description:** Fast Walsh-Hadamard Transform of array  $a$ :  $fwht(a) = (\sum_i (-1)^{pc(i \& 0)} a_i, \dots, \sum_i (-1)^{pc(i \& n-1)} a_i)$ . One can use it to do xor-convolution. Length of  $a$  should be a power of 2.

**Time:**  $\mathcal{O}(N \log N)$ , where  $N$  is the length of  $a$ .

15 lines

```

template<class T>
void fwht(vector<T> &a, int is_inv) {
    int n = sz(a);
    for (int s = 1; s < n; s <= 1)
        for (int i = 0; i < n; i += s << 1)
            rep(j, 0, s - 1) {
                T x = a[i + j], y = a[i + j + s];
                a[i + j] = x + y;
                a[i + j + s] = x - y;
            }
    if (is_inv) {
        for (auto &x: a) x = x / n;
    }
} // hash-cpp-all = 69be2c88185ff1254f92dea3f228137e

```

### fwht-eval.cpp

**Description:** Let  $b = fwt(a)$ . One can calculate  $b_{id}$  for some index  $id$  in  $\mathcal{O}(N)$  time. Length of  $a$  should be a power of 2.

**Time:**  $\mathcal{O}(N)$ , where  $N$  is the length of  $a$ .

10 lines

```

template<class T>
T fwt_eval(const vector<T> &a, int id) {
    int n = sz(a);
    T res = 0;
    rep(i, 0, n - 1) {
        if (__builtin_popcount(i & id) & 1) res -= a[i];
        else res += a[i];
    }
    return res;
} // hash-cpp-all = 3803dcab58e34af9decd2a3be78a5724

```

## 6.2 Linear Systems

### matrix.cpp

**Description:** Matrix struct.  $Gaussian(C)$  eliminates the first  $C$  columns and returns the rank of matrix induced by first  $C$  columns.  $inverse()$  gives the inverse of the matrix.  $SolveLinear(A, b)$  solves linear system  $Ax = b$  for matrix  $A$  and vector  $b$ . Besides, you need function  $isZero$  for your template  $T$ .

**Usage:** For  $SolveLinear()$ :

```

bool isZero(double x) { return abs(x) <= 1e-9; } // global
Matrix<double> A(3, 4);
vector<double> b(3);
... // set values for A and b.
vector<double> xs = SolveLinear(A, b);

```

**Time:**  $\mathcal{O}(nm \min\{n, m\})$  for Gaussian, inverse and  $SolveLinear$ . 98 lines

```

template<class T>
struct Matrix {
    using Mat = Matrix; // hash-cpp-1
    using Vec = vector<T>;

```

```

    vector<Vec> a;

```

```

    Matrix(int n, int m) {
        assert(n > 0 && m > 0);
        a.assign(n, Vec(m));
    }
    Matrix(const vector<Vec> &a): a(a) {
        assert(sz(a) > 0 && sz(a[0]) > 0);
    }

```

```

    Vec& operator [] (int i) const { return (Vec&) a[i]; }
    // hash-cpp-1 = 273826412c0415697d0c90ccf0130f7c

    Mat operator +(const Mat &b) const {
        int n = sz(a), m = sz(a[0]);
        Mat c(n, m);
        rep(i, 0, n - 1) rep(j, 0, m - 1) c[i][j] = a[i][j] + b
            ↪ [i][j];
        return c;
    }

    Mat operator -(const Mat &b) const {
        int n = sz(a), m = sz(a[0]);
        Mat c(n, m);
        rep(i, 0, n - 1) rep(j, 0, m - 1) c[i][j] = a[i][j] - b
            ↪ [i][j];
        return c;
    }

    Mat operator *(const Mat &b) const {
        int n = sz(a), m = sz(a[0]), l = sz(b[0]);
        assert(m == sz(b.a));
        Mat c(n, l);
        rep(i, 0, n - 1) rep(k, 0, m - 1) rep(j, 0, l - 1) c[i]
            ↪ [j] += a[i][k] * b[k][j];
        return c;
    }

    Mat tran() const {
        int n = sz(a), m = sz(a[0]);
        Mat res(m, n);
        rep(i, 0, n - 1) rep(j, 0, m - 1) res[j][i] = a[i][j];
        return res;
    }

    // Eliminate the first C columns, return the rank of
    ↪ matrix induced by first C columns.
    int Gaussian(int C) { // hash-cpp-2
        int n = sz(a), m = sz(a[0]), rk = 0;
        assert(C <= m);
        rep(c, 0, C - 1) {
            int id = rk;
            while (id < n && ::isZero(a[id][c])) id++;
            if (id == n) continue;
            if (id != rk) swap(a[id], a[rk]);

            T tmp = a[rk][c];
            for (auto &x: a[rk]) x /= tmp;
            rep(i, 0, n - 1) if (i != rk) {
                T fac = a[i][c];
                rep(j, 0, m - 1) a[i][j] -= fac * a[rk][j];
            }
            rk++;
        }
        return rk;
    } // hash-cpp-2 = 1d0d00b2e87f9e2d7abb939d59db1202

    Mat inverse() const { // hash-cpp-3
        int n = sz(a), m = sz(a[0]);
        assert(n == m);
        auto b = *this;

        rep(i, 0, n - 1) b[i].resize(n * 2, 0), b[i][n + i] =
            ↪ 1;
        assert(b.Gaussian(n) == n);
        for (auto &row: b.a) row.erase(row.begin(), row.begin()
            ↪ + n);

```

```

        return b;
    } // hash-cpp-3 = 7f21877d9ac6d76d755d6b79b03be029

    friend pair<bool, Vec> SolveLinear(Mat A, const Vec &b) {
        ↪ // hash-cpp-4
        int n = sz(A.a), m = sz(A[0]);
        assert(sz(b) == n);
        rep(i, 0, n - 1) A[i].push_back(b[i]);
        int rk = A.Gaussian(m);
        rep(i, rk, n - 1) if (::isZero(A[i].back()) == 0)
            ↪ return {0, Vec{}};
        Vec res(m);
        revrep(i, 0, rk - 1) {
            T x = A[i][m];
            int last = -1;
            revrep(j, 0, m - 1) if (::isZero(A[i][j]) == 0) {
                x -= A[i][j] * res[j];
                last = j;
            }
            if (last != -1) res[last] = x;
        }
        return {1, res};
    } // hash-cpp-4 = ca7ea2663b271d600d1d50cb6367eb72
};

```

### linear-base.cpp

**Description:** Maximum weighted of Linear Base of vector space  $\mathbb{Z}_2^d$ .  $T$  is the type of vectors and  $Z$  is the type of weights.  $w[i]$  is the non-negative weight of  $a[i]$ . Keep  $w[]$  zero to use unweighted Linear Base. **Time:**  $\mathcal{O}(d \cdot \frac{d}{w})$  for insert;  $\mathcal{O}(d^2 \cdot \frac{d}{w})$  for union;  $\mathcal{O}(d \cdot \frac{d}{w})$  for  $kth()$ . 16 lines

```

template<int d, class T = bitset<d>, class Z = int>
struct LB {
    vector<T> a; // hash-cpp-1
    vector<Z> w;

    T& operator [] (int i) const { return (T&)a[i]; }
    LB(): a(d), w(d) {}

    // insert x. return 1 if the base is expanded.
    int insert(T x, Z val = 0) {
        revrep(i, 0, d - 1) if (x[i]) {
            if (a[i] == 0) {
                a[i] = x;
                w[i] = val;
                return 1;
            } else if (val > w[i]) {
                swap(a[i], x);
                swap(w[i], val);
            }
            x ^= a[i];
        }
        return 0;
    } // hash-cpp-1 = 18f5fb93fd62247833ec8b725ab4e689

    // View vecotrs as binary numbers. Then calculate the
    ↪ minimum number we can get if we add vectors from
    ↪ linear base (with weight at least $val$) to $x$.
    T ask_min(T x, Z val = 0) { // hash-cpp-2
        revrep(i, 0, d - 1) {
            if (x[i] && w[i] >= val) x ^= a[i]; // change x[i] to
            ↪ x[i] == 0 to ask maximum value we can get.
        }
        return x;
    } // hash-cpp-2 = 2abeaf37e03b3f853b1ccea025ec88ef

    // Compute the union of two bases.

```

```

    friend LB operator +(LB a, const LB &b) { // hash-cpp-3
        rep(i, 0, d - 1) if (b[i] != 0) a.insert(b[i]);
        return a;
    } // hash-cpp-3 = 9e0a459d8f20e3374e28ffb59a38c89e

    // Returns the k-th smallest number spanned by vectors of
    ↪ weight at least $val$. k starts from 0.
    T kth(unsigned long long k, Z val = 0) { // hash-cpp-4
        int N = 0;
        rep(i, 0, d - 1) N += (a[i] != 0 && w[i] >= val);
        if (k >= (1ull << N)) return -1; // return -1 if k is
        ↪ too large.
        T res = 0;
        revrep(i, 0, d - 1) if (a[i] != 0 && w[i] >= val) {
            --N;
            auto bit = k >> N & 1;
            if (res[i] != bit) res ^= a[i];
        }
        return res;
    } // hash-cpp-4 = 3d8a0ecfd6a4e4f5ad30dafc3e1b6379
};

```

### linear-base-intersect.cpp

**Description:** Intersection of two unweighted linear bases.  $T$  should be of length at least  $2d$ . **Time:**  $\mathcal{O}(d^2 \cdot \frac{d}{w})$ . 16 lines

```

"linear-base.cpp"
template<int d, class T = bitset<d * 2>>
LB<d, T> intersect(LB<d, T> a, const LB<d, T> &b) {
    LB<d, T> res;
    rep(i, 0, d - 1) if (a[i] != 0) a[i][d + i] = 1;
    T msk(string(d, '1'));
    rep(i, 0, d - 1) {
        T x = a.ask_min(b[i]);
        if ((x & msk) != 0) a.insert(x);
        else {
            T y = 0;
            rep(j, 0, d - 1) if (x[d + j]) y ^= a[j];
            res.insert(y & msk);
        }
    }
    return res;
} // hash-cpp-all = ef800af439fc0dc8b3438fa8b7a8af86

```

### Z3-vector.cpp

**Description:** vector in  $\mathbb{Z}_3$ . **Time:**  $\mathcal{O}(d/w)$  for +, -, \* and /. 45 lines

```

template<int d>
struct v3 {
    bitset<d> a[3]; // hash-cpp-1

    v3() { a[0].set(); }

    void set(int pos, int x) {
        rep(i, 0, 2) a[i][pos] = (i == x);
    }
    int operator [] (int i) const {
        if (a[0][i]) return 0;
        else if (a[1][i]) return 1;
        else return 2;
    }
    v3 operator +(const v3 &rhs) const {
        v3 res;
        res.a[0] = (a[0] & rhs.a[0]) | (a[1] & rhs.a[2]) | (a
            ↪ [2] & rhs.a[1]);

```

```

    res.a[1] = (a[0] & rhs.a[1]) | (a[1] & rhs.a[0]) | (a
        ↪ [2] & rhs.a[2]);
    res.a[2] = (~res.a[0] & ~res.a[1]);
    return res;
}
v3 operator -(const v3 &rhs) const {
    v3 tmp = rhs;
    swap(tmp.a[1], tmp.a[2]);
    return *this + tmp;
}
v3 operator *(int rhs) const {
    if (rhs % 3 == 0) return v3{};
    else {
        auto res = *this;
        if (rhs % 3 == 2) swap(res.a[1], res.a[2]);
        return res;
    }
}
v3 operator /(int rhs) const {
    assert(rhs % 3 != 0);
    return *this * rhs;
} // hash-cpp-1 = 0d5a33ef7c028d641716f6f8a1ebf1b5

friend string to_string(const v3 &a) {
    string s;
    rep(i, 0, d - 1) s.push_back('0' + a[i]);
    return s;
}
};

```

### simplex.cpp

**Description:** Solves a general linear maximization problem: maximize  $c^T x$  subject to  $Ax \leq b$ ,  $x \geq 0$ . Returns  $\{res, x\}$ :  $res = 0$  if the program is infeasible;  $res = 1$  if there exists an optimal solution;  $res = 2$  if the program is unbounded.  $x$  is valid only when  $res = 1$ .  $T$  can be **double** or **long double**.

**Time:**  $\mathcal{O}(NM * \#pivots)$ , where  $N$  is the number of constraints and  $M$  is the number of variables.

72 lines

```

template<class T>
pair<int, vector<T>> Simplex(const vector<vector<T>> &A,
    ↪ const vector<T> &b, const vector<T> &c) {
    const T eps = 1e-8;

    assert(sz(A) > 0 && sz(A[0]) > 0);
    int n = sz(A);
    int m = sz(A[0]);
    vector<vector<T>> a(n + 1, vector<T>(m + 1));
    rep(i, 0, n - 1) rep(j, 0, m - 1) a[i + 1][j + 1] = A[i][
        ↪ j];
    rep(i, 0, n - 1) a[i + 1][0] = b[i];
    rep(j, 0, m - 1) a[0][j + 1] = c[j];

    vi left(n + 1, up(m + 1));
    iota(all(left), m);
    iota(all(up), 0);

    auto pivot = [&](int x, int y) {
        swap(left[x], up[y]);
        T k = a[x][y];
        a[x][y] = 1;
        vi pos;
        rep(j, 0, m) {
            a[x][j] /= k;
            if (fabs(a[x][j]) > eps) pos.push_back(j);
        }
        rep(i, 0, n) {

```

```

            if (fabs(a[i][y]) < eps || i == x) continue;

            k = a[i][y];
            a[i][y] = 0;
            for (int j : pos) a[i][j] -= k * a[x][j];
        }
    };

    while (1) {
        int x = -1;
        rep(i, 1, n) if (a[i][0] < -eps && (x == -1 || a[i][0]
            ↪ < a[x][0])) {
            x = i;
        }
        if (x == -1) break;

        int y = -1;
        rep(j, 1, m) if (a[x][j] < -eps && (y == -1 || a[x][j]
            ↪ < a[x][y])) {
            y = j;
        }
        if (y == -1) return {0, vector<T>{}}; // infeasible
        pivot(x, y);
    }

    while (1) {
        int y = -1;
        rep(j, 1, m) if (a[0][j] > eps && (y == -1 || a[0][j] >
            ↪ a[0][y])) {
            y = j;
        }
        if (y == -1) break;

        int x = -1;
        rep(i, 1, n) if (a[i][y] > eps && (x == -1 || a[i][0] /
            ↪ a[i][y] < a[x][0] / a[x][y])) {
            x = i;
        }
        if (x == -1) return {2, vector<T>{}}; // unbounded
        pivot(x, y);
    }

    vector<T> ans(m);
    rep(i, 1, n) {
        if (1 <= left[i] && left[i] <= m) {
            ans[left[i] - 1] = a[i][0];
        }
    }
    return {1, ans};
} // hash-cpp-all = 1b84e92f161dc13c0d93359656b5b636

```

### matroid-intersection.cpp

**Description:** Given a ground set  $E$  and two matroid  $M_1 = (E, I_1)$  and  $M_2 = (E, I_2)$ , compute a largest independent set in their intersection  $M = (E, I_1 \cap I_2)$ , i.e. an element in  $I_1 \cap I_2$  of largest size. Denote by  $as$  the ground set.  $rebuild(A)$  rebuilds the data structure using elements in  $A$ . Then  $check1(x)$  returns if  $A \cup \{x\} \in I_1$  and  $check2$  returns if  $A \cup \{x\} \in I_2$  using the data structure just built before.

**Time:**  $\mathcal{O}(r^2|E|)$ , where  $r = \min(r(E, I_1), r(E, I_2))$ .

56 lines

```

template<class T>
vector<T> MatroidIntersection(const vector<T> &as, function
    ↪ void(const vector<T> &) > rebuild, function<bool(const
    ↪ T &) > check1, function<bool(const T &) > check2) {
    int n = sz(as);
    vi used(n);
    vvi g;

```

```

vector<T> A;

auto augment = [&]() {
    int tot = n, s = tot++, t = tot++;
    g.assign(tot, {});
    A.clear();
    rep(i, 0, n - 1) if (used[i]) A.push_back(as[i]);
    rebuild(A);

    rep(y, 0, n - 1) if (used[y] == 0) {
        int cnt = 0;
        if (check1(as[y])) g[s].push_back(y), cnt++;
        if (check2(as[y])) g[y].push_back(t), cnt++;
        if (cnt == 2) { // if we have s -> y -> t, then we
            ↪ could just augment via this path!
            used[y] = 1;
            return 1;
        }
    }

    rep(x, 0, n - 1) if (used[x]) {
        A.clear();
        rep(i, 0, n - 1) if (used[i] && i != x) A.push_back(
            ↪ as[i]);
        rebuild(A);
        rep(y, 0, n - 1) if (used[y] == 0) {
            if (check1(as[y])) g[x].push_back(y);
            if (check2(as[y])) g[y].push_back(x);
        }
    }

    vi dis(tot, -1), pre(tot);
    vi que{s};
    dis[s] = 0;
    rep(ind, 0, sz(que) - 1) {
        int now = que[ind];
        for (auto v: g[now]) if (dis[v] == -1) {
            dis[v] = dis[now] + 1;
            que.push_back(v);
            pre[v] = now;
        }
    }
    if (dis[t] == -1) return 0;
    int now = pre[t];
    while (now != s) {
        used[now] ^= 1;
        now = pre[now];
    }
    return 1;
};

while (augment());
vector<T> res;
rep(i, 0, n - 1) if (used[i]) res.push_back(as[i]);
return res;
} // hash-cpp-all = 1fe250370d9628e34d6167963bce2cb6

```

## 6.3 Functions

### integrate.cpp

**Description:** Let  $f(x)$  be a continuous function over  $[a, b]$  and have a fourth derivative,  $f^{(4)}(x)$ , over this interval. If  $M$  is the maximum value of  $|f^{(4)}(x)|$  over  $[a, b]$ , then the upper bound for the error is  $O\left(\frac{M(b-a)^5}{N^4}\right)$ .

**Time:**  $\mathcal{O}(N \cdot T)$ , where  $T$  is the time for evaluating  $f$  once.

9 lines

```

template<class T = double>
T SimpsonsRule(const function<T(T)> &f, T a, T b, int N =
    ↪ 1000) {

```

```

T res = 0;
T h = (b - a) / (N * 2);
res += f(a);
res += f(b);
rep(i, 1, N * 2 - 1) res += f(a + h * i) * (i & 1 ? 4 :
    ↪2);
return res * h / 3;
} // hash-cpp-all = defd8926ebf2de40cd1a9e5dc26385c3

```

integrate-adaptive.cpp

**Description:** Adaptive Simpson's Rule. It is somehow necessary to set the minimum depth of recursion. We use *dep* here. Change it smaller if Time Limit is tight.

```

template<class T = double>
T AdaptiveIntegrate(const function<T(T)>& f, T a, T b, T
    ↪eps = 1e-8, int dep = 5) {
    auto simpson = [&](T a, T b) {
        T c = (a + b) / 2;
        return (f(a) + f(c) * 4 + f(b)) * (b - a) / 6;
    };
    auto rec = [&](auto &dfs, T a, T b, T eps, T S, int dep)
        ↪-> T {
        T c = (a + b) / 2;
        T S1 = simpson(a, c), S2 = simpson(c, b), sum = S1 + S2
            ↪;
        if ((abs(sum - S) <= 15 * eps || b - a < 1e-10) && dep
            ↪<= 0) return sum + (sum - S) / 15;
        return dfs(dfs, a, c, eps / 2, S1, dep - 1) + dfs(dfs,
            ↪c, b, eps / 2, S2, dep - 1);
    };
    return rec(rec, a, b, eps, simpson(a, b), dep);
} // hash-cpp-all = c36fe3593b4c741c0e951ea53c574edd

```

recursive-ternary-search.cpp

**Description:** For convex function  $f: \mathbb{R}^d \rightarrow \mathbb{R}$ , we can approximately find the global minimum using ternary search on each coordinate recursively.  $d$  is the dimension;  $mn, mx$  record the minimum and maximum possible value of each coordinate (the region you do ternary search);  $f$  is the convex function.  $T$  can be **double** or **long double**.

**Time:**  $\mathcal{O}\left(\log(1/\epsilon)^d \cdot C\right)$ , where  $C$  is the time for evaluating the function  $f$ .

```

template<class T> T RecTS(int d, const vector<T>& mn, const
    ↪ vector<T>& mx, function<T(const vector<T>&)> f) {
    vector<T> xs(d);
    auto dfs = [&](auto &dfs, int dep) {
        if (dep == d) return f(xs);
        T l = mn[dep], r = mx[dep];
        rep(_, 1, 60) { // change here if time is tight.
            T m1 = (l * 2 + r) / 3;
            T m2 = (l + r * 2) / 3;

            xs[dep] = m1; T res1 = dfs(dfs, dep + 1);
            xs[dep] = m2; T res2 = dfs(dfs, dep + 1);
            if (res1 < res2) r = m2;
            else l = m1;
        }
        xs[dep] = (l + r) / 2;
        return dfs(dfs, dep + 1);
    };
    return dfs(dfs, 0);
} // hash-cpp-all = cf72be7d40cc4f7693a87647aae4e6b4

```

modnum.cpp

**Description:** Modular integer with  $mod \leq 2^{30} - 1$ . Note that there are several advantages to use this code: 1. You do not need to keep writing  $\%mod$ ; 2. It is good to use this struct when doing Gaussian Elimination / Fast Walsh-Hadamard Transform; 3. Sometimes the input number is greater than  $mod$  and this code handles it. Do not write things like  $Mint\{1/3\}.pow(10)$  since  $1/3$  simply equals 0. Do not write things like  $Mint\{a * b\}$  where  $a$  and  $b$  are int since you might first have integer overflow.

**Usage:** Define the followings globally:  
const int mod = 998244353;  
using Mint = Z<mod>;

```

template<const int &mod>
struct Z {
    // hash-cpp-1
    int x;
    Z(ll a = 0): x(a % mod) { if (x < 0) x += mod; }
    explicit operator int() const { return x; }

    Z& operator +=(Z b) { x += b.x; if (x >= mod) x -= mod;
        ↪return *this; }
    Z& operator -=(Z b) { x -= b.x; if (x < 0) x += mod;
        ↪return *this; }
    Z& operator *=(Z b) { x = 1ll * x * b.x % mod; return *
        ↪this; }
    friend Z operator +(Z a, Z b) { return a + b; }
    friend Z operator -(Z a, Z b) { return a - b; }
    friend Z operator *(Z a, Z b) { return a * b; }
    // hash-cpp-1 = e5f2469d533a39d2945e75688e0b7e94

    // the followings are for ntt and polynomials.
    Z pow(ll k) const { // hash-cpp-2
        Z res = 1, a = *this;
        for (; k >= 1, a = a * a) if (k & 1) res = res * a;
        return res;
    }
    Z& operator /=(Z b) {
        assert(b.x != 0);
        return *this *= b.pow(mod - 2);
    }
    friend Z operator /(Z a, Z b) { return a /= b; }

    static int getMod() { return mod; } // ntt need this.
    // hash-cpp-2 = 25825dd33306e07c0d0faf87a0e74882

    friend string to_string(Z a) { return to_string(a.x); }
};

```

euclidean.cpp

**Description:** Compute  $\sum_{i=1}^n \lfloor \frac{ai+b}{c} \rfloor$  for integer numbers  $a, b, c, n$ .  
**Time:**  $\mathcal{O}(\log c)$ .

```

template<class T>
T Euclidean(ll a, ll b, ll c, ll n) {
    T res = 0;
    if (a >= c || b >= c) {
        res += T(a / c) * n * (n + 1) / 2;
        res += T(b / c) * (n + 1);
        a %= c;
        b %= c;
    }
    if (a != 0) {
        ll m = ((__int128)a * n + b) / c;
        res += T(m) * n - Euclidean<T>(c, c - b - 1, a, m - 1);
    }
    return res;
}

```

```

} // hash-cpp-all = 05c2bd1a556cb8149508fe555ca3d3f5

```

exgcd.cpp

**Description:** Solve the integer equation  $ax + by = \gcd(a, b)$  for  $a, b \geq 0$  and returns  $x$  and  $y$  such that  $|x| \leq b$  and  $|y| \leq a$ . **Note that** retured value  $x$  and  $y$  are not guaranteed to be positive!  
**Time:**  $\mathcal{O}(\log \max\{a, b\})$ .

```

template<class T>
pair<T, T> exgcd(T a, T b) {
    if (b == 0) return {1, 0};
    auto [x, y] = exgcd(b, a % b);
    return {y, x - a / b * y};
} // hash-cpp-all = f1ae06792ef3524ec6f5aff196c54a51

```

chinese.cpp

**Description:** Chinese Remainder Theorem for solveing equations  $x \equiv a_i(mod\ m_i)$  for  $i = 0, 1, \dots, n - 1$  such that all  $m_i$ -s are pairwise-coprime. Returns  $a$  such  $x = a + k \cdot (\prod_{i=0}^{n-1} m_i)$ ,  $k \in \mathbb{Z}$  are solutions. **Note that** you need to choose type  $T$  to fit  $(\prod_i m_i) \cdot (\max_i m_i)$ .

**Time:**  $\mathcal{O}\left(n \log(\prod_{i=0}^{n-1} m_i)\right)$ .

```

template<class T>
T CRT(const vector<T>& as, const vector<T>& ms) {
    T M = 1, res = 0;
    for (auto x: ms) M *= x;
    rep(i, 0, sz(as) - 1) {
        T m = ms[i], Mi = M / m;
        auto [x, y] = exgcd(Mi, m);
        res = (res + as[i] % m * Mi * x) % M;
    }
    return (res + M) % M;
} // hash-cpp-all = 617e5d398d307d9d9399aff7908ae7ed

```

chinese-common.py

```

# Author: Yuhao Yao
# Date: 22-10-24
def exgcd(a, b):
    if b == 0:
        return 1, 0
    x, y = exgcd(b, a % b)
    return y, x - a // b * y

# Returned A is the minimum non-negative integer satisfying
    ↪ given two equations.
def merge(al, ml, a2, m2):
    if ml == -1 or m2 == -1:
        return -1, -1
    y1, y2 = exgcd(ml, m2)
    g = ml * y1 + m2 * y2
    if (a2 - al) % g != 0:
        return -1, -1
    y1 = y1 * ((a2 - al) // g) % (m2 // g)
    if y1 < 0:
        y1 += m2 // g
    M = ml // g * m2
    A = ml * y1 + al
    return A, M

# Given a list of pairs (a_i, m_i) representing equations x
    ↪ = a_i (mod m_i)
# Return a, m such that a + m * k are solutions. -1, -1
    ↪ means that there is no solution.
def general_chinese(ps):
    a, m = 0, 1

```

```
for a2, m2 in ps:
    a, m = merge(a, m, a2, m2)
return a, m
```

factorization.cpp

**Description:** Primality test and Fast Factorization. The *mul* function supports  $0 \leq a, b < c < 7.268 \times 10^{18}$  and is a little bit faster than `__int128`.

**Time:**  $\mathcal{O}(x^{1/4})$  for pollard-rho and same for factorizing *x*. 64 lines

```
namespace Factorization {
    inline ll mul(ll a, ll b, ll c) { // hash-cpp-1
        ll s = a * b - c * ll((long double) a / c * b + 0.5);
        return s < 0 ? s + c : s;
    }

    ll mPow(ll a, ll k, ll mod) {
        ll res = 1;
        for (; k >= 1, a = mul(a, a, mod)) if (k & 1) res =
            ↪ mul(res, a, mod);
        return res;
    }

    bool miller(ll n) {
        auto test = [&](ll n, int a) {
            if (n == a) return true;
            if (n % 2 == 0) return false;

            ll d = (n - 1) >> __builtin_ctzll(n - 1);
            ll r = mPow(a, d, n);
            while (d < n - 1 && r != 1 && r != n - 1) {
                d <<= 1;
                r = mul(r, r, n);
            }
            return r == n - 1 || d & 1;
        };

        if (n == 2) return 1;
        for (auto p: vi{2, 3, 5, 7, 11, 13}) if (test(n, p) ==
            ↪ 0) return 0;
        return 1;
    } // hash-cpp-1 = bb239644542d955fdb24ad66508e26d6

    mt19937_64 rng(chrono::steady_clock::now().
        ↪ time_since_epoch().count()); // hash-cpp-2
    ll myrand(ll a, ll b) { return uniform_int_distribution<
        ↪ ll>(a, b)(rng); }

    ll pollard(ll n) { // return some nontrivial factor of n.
        auto f = [&](ll x) { return ((__int128) x * x + 1) % n;
            ↪ };

        ll x = 0, y = 0, t = 30, prd = 2;
        while (t++ % 40 || gcd(prd, n) == 1) {
            // speedup: don't take __gcd in each iteration.
            if (x == y) x = myrand(2, n - 1), y = f(x);
            ll tmp = mul(prd, abs(x - y), n);
            if (tmp) prd = tmp;
            x = f(x), y = f(f(y));
        }
        return gcd(prd, n);
    }

    vector<ll> factorize(ll n) {
        vector<ll> res;

        auto dfs = [&](auto &dfs, ll x) {
```

```
        if (x == 1) return;
        if (miller(x)) res.push_back(x);
        else {
            ll d = pollard(x);
            dfs(dfs, d);
            dfs(dfs, x / d);
        }
    };
    dfs(dfs, n);
    return res;
} // hash-cpp-2 = 11aa8a52e6d3fb6ce4aa98100d100a3c
}
```

is-prime.cpp 1 lines

```
// TODO
```

cont-frac.cpp 1 lines

```
// TODO
```

adleman-manders-miller.cpp 1 lines

```
// TODO
```

discrete-log.cpp 1 lines

```
// TODO
```

sieve.cpp

**Description:** Sieve for prime numbers / multiplicative functions in  $\{1, 2, \dots, N\}$  in linear time.

**Time:**  $\mathcal{O}(N)$ . 33 lines

```
struct LinearSieve {
    vi ps, minp;
    vi d, facnum, phi, mu;
    LinearSieve(int n): minp(n + 1), d(n + 1), facnum(n + 1),
        ↪ phi(n + 1), mu(n + 1) {
        facnum[1] = phi[1] = mu[1] = 1;
        rep(i, 2, n) {
            if (minp[i] == 0) {
                ps.push_back(i);
                minp[i] = i;
                d[i] = 1;
                facnum[i] = 2;
                phi[i] = i - 1;
                mu[i] = -1;
            }
            for (auto p: ps) {
                ll v = ll1 * i * p;
                if (v > n) break;
                minp[v] = p;
                if (i % p == 0) {
                    d[v] = d[i] + 1;
                    facnum[v] = facnum[i] / (d[i] + 1) * (d[v] + 1);
                    phi[v] = phi[i] * p;
                    mu[v] = 0;
                    break;
                }
                d[v] = 1;
                facnum[v] = facnum[i] * 2;
                phi[v] = phi[i] * (p - 1);
                mu[v] = -mu[i];
            }
        }
    }
};
```

```
}; // hash-cpp-all = 496b1c3a9df8a550e6022a4573bb36dd
```