Eidgenössische Technische Hochschule Zürich

lETHargy

Antti Röyskö, Yuhao Yao, Marcel Bezdrighin

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

2022-10-18

Contest (1)

template.cpp

7 lines

```
#include <bits/stdc++.h>
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(x) x.begin(), x.end()
#define sz(x) (int)(x).size()
using pii = pair<int, int>
using vi = vector<int>
using ll = long long;
```

hash.sh

1 lines

```
tr -d '[:space:]' | md5sum
```

hash-cpp.sh

1 lines

```
cpp -dD -P -fpreprocessed | tr -d '[:space:]' | md5sum
```

1.1 Notes

1.1.1 Vscode config

"editor.insertSpaces": false,
"window.titleBarStyle": "custom",
"window.customMenuBarAltFocus": false,

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

1.1.2 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

```
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
template<class T>
T rand(T a, T b) { return uniform_int_distribution<T>(a, b)(rng); }
template<class T>
T rand() { return uniform_int_distribution<T>()(rng); }
// shuffle(perm.begin(), perm.end(), rng);
```

hilbert-mos.cpp

Description: Hilbert curve sorting order for Mo's algorithm. Sorts queries (L_i, R_i) where $0 \le L_i \le R_i < n$ into order π, such that $\sum_i \left|L_{\pi_{i+1}} - L_{\pi_i}\right| + \left|R_{\pi_{i+1}} - R_{\pi_i}\right| = \mathcal{O}(n\sqrt{q})$

Usage: hilbertOrder(n, qs) returns π

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

21 lines

```
ll hilbertOrd(int y, int x, int h) {
  if (h == -1) return 0;
  int s = (1 << h), r = (1 << h) - 1;
  int y0 = y >> h, x0 = x >> h;
  int y1 = y & r, x1 = x & r;
  int ny = (y0 ? y1 : (x0 ? r - x1 : x1)); // x1 : r - x1));
  int nx = (y0 ? x1 : (x0 ? r - y1 : y1)); // y1 : r - y1)); // r - y1 : y1));
  return s*s * (2*x0 + (x0 ^ y0)) + hilbertOrd(ny, nx, h-1);
}
vector<int> hilbertOrder(int n, const vector<pair<int, int>>& qs) {
  int h = 0, q = qs.size();
  while((1 << h) < n) ++h;

  vector<pair<ll, int>> tmp(q);
  for (int i = 0; i < q; ++i) tmp[i] = {hilbertOrd(qs[i].first, qs[i].second, h - 1), i};
  sort(tmp.begin(), tmp.end());

  vector<int> res(q);
  for (int qi = 0; qi < q; ++qi) res[qi] = tmp[qi].second;
  return res;
} // hash-cpp-all = 6467dd464ea41a6009895a50f6f12523
```

Data structure (3)

fenwick.cpp

Description: Fenwick tree with built in binary search. Can be used as a indexed set.

Usage: ??

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

35 lines

```
class Fenwick {
  private:
    vector<ll> val;
  public:
    Fenwick(int n) : val(n+1, 0) {}

    // Adds v to index i
    void add(int i, ll v) {
      for (++i; i < val.size(); i += i & -i) {
        val[i] += v;
      }
    }

    // Calculates prefix sum up to index i
    ll get(int i) {
      ll res = 0;
      for (++i; i > 0; i -= i & -i) {
        res += val[i];
      }
      return res;
    }
    ll get(int a, int b) { return get(b) - get(a-1); }

    // Assuming prefix sums are non-decreasing, finds last i s.t. get(i) <= v
    int search(ll v) {
      int res = 0;
      for (int h = 1<<30; h; h >>= 1) {
        if ((res | h) < val.size() && val[res | h] <= v) {
          res |= h;
          v -= val[res];
        }
      }
      return res - 1;
    }
}; // hash-cpp-all = 0d390772acaff4360d0f4d76da45148e
```

segtree.cpp

Description: Segment tree supporting range addition and range sum, minimum queries

Usage: ??

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

58 lines

```
// Segment tree for range addition, range sum and range minimum.
class SegTree {
  private:
    vector<ll> sum, minv, tag;
    int h = 1;

    // Returns length of interval corresponding to position i
    ll len(int i) { return h >> (31 - __builtin_clz(i)); }

    void apply(int i, ll v) {
      sum[i] += v * len(i);
      minv[i] += v;
      if (i < h) tag[i] += v;
    }
    void push(int i) {
      if (tag[i] == 0) return;
      apply(2*i, tag[i]);
      apply(2*i+1, tag[i]);
      tag[i] = 0;
    }

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

// O(n) preprocessing, 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 <= len <= 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
```

sparse-table.cpp

Description: Sparse Table.
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> &a, const F &f): n(sz(a)),
    ↪func(f) {
    assert(n > 0);
    st.assign(__lg(n) + 1, vector<T>(n));
    st[0] = a;
    rep(i, 1, __lg(n)) rep(j, 0, n - (1 << i)) st[i][j] =
      ↪func(st[i - 1][j], st[i - 1][j + (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 = b743d83364ed3febf454197dd9d6aa63
```

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();) {
        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 = 51ad9045bff4d74f5c7b851530e02304
```

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-prique.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. Point apply and thus no lazy propogation.
Usage: Always define a global apply function to tell segment tree how you apply modification.
Combine is set as plus so if you just let T be numerical type then you have range sum in the info and as range query result. To have something different, say rangeMin, define a struct with constructer and + operation.
Time: $\mathcal{O}(\log N)$ per operation.

61 lines

```
template<class Info> class PersistSegtree {
// hash-cpp-1
  struct node { Info info; int ls, rs; };
  int n;
  vector<node> t;
public:
  // 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;
      ::apply(t[i].info, val...);
      if (l == r) 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);
    };
    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 = 920335506780ce4054d72e2496d81e6c

  // 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
};
```

2d-segtree.cpp

Description: 2D segment tree. Point apply and thus no lazy propogation.
Usage: Always define a global apply function to tell segment tree how you apply modification.
Combine is set as plus so if you just let T be numerical type then you have range sum in the info and as range query result. To have something different, say rangeMin, define a struct with constructer and + operation.
Time: $\mathcal{O}(\log^2 N)$ per operation.

78 lines

```
template<class T> struct SegTree2D {
  struct iNode { T 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> inner;
  vector<oNode> outer;

  // 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), inner(1), outer(1) {
    inner.reserve(q * (__lg(oR - oL + 1) + 2) * (__lg(iR -
      ↪iL + 1) + 2) + 1);
    outer.reserve(q * (__lg(oR - oL + 1) + 2) + 1);
  }

  int newInner() { inner.push_back({}); return sz(inner) -
    ↪1; }
  int newOuter() { outer.push_back({}); return sz(outer) -
    ↪1; }

  void pull(int i) {
    auto &[info, ls, rs] = inner[i];
    info = inner[ls].info + inner[rs].info;
  }
  void apply(int i, const T &val) {
    ::apply(inner[i].info, val);
  }

  // return new root id.
  int pointApply(int rt, int op, int ip, const T &val) {
    auto idfs = [&](auto dfs, int &i, int l, int r) {
      if (!i) i = newInner();
      if (l == r) {
        apply(i, val);
        return;
      }
      int mid = (l + r) >> 1;
      if (ip <= mid) dfs(dfs, inner[i].ls, l, mid);
      else dfs(dfs, inner[i].rs, mid + 1, r);
      pull(i);
    };
    auto odfs = [&](auto dfs, int &i, int l, int r) {
      if (!i) i = newOuter();
      idfs(idfs, outer[i].id, iL, iR);
      if (l == r) return;
      int mid = (l + r) >> 1;
      if (op <= mid) dfs(dfs, outer[i].ls, l, mid);
      else dfs(dfs, outer[i].rs, mid + 1, r);
      return;
    };
    odfs(odfs, rt, oL, oR);
    return rt;
  }

  T rangeAsk(int rt, int qol, int qor, int qil, int qir) {
    T 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 + inner[i].info;
        return;
      }
      int mid = (l + r) >> 1;
      dfs(dfs, inner[i].ls, l, mid);
      dfs(dfs, inner[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, outer[i].id, iL, iR);
        return;
      }
      int mid = (l + r) >> 1;
      dfs(dfs, outer[i].ls, l, mid);
      dfs(dfs, outer[i].rs, mid + 1, r);
```

```
    };
    odfs(odfs, rt, oL, oR);
    return res;
  }
}; // hash-cpp-all = 8cacae47df103b5a46ee857150a26646
```

pq-tree.cpp
1 lines

```
// TODO
```

treap.cpp
1 lines

```
// TODO
```

matrix-seg.cpp
1 lines

```
// TODO: segment tree for historic information
```

Graph algorithms (4)

dinic.cpp

Description: Dinic algorithm for flow graph $G = (V, E)$.
Usage: Always run $MaxFlow(src, sink)$ for some src and $sink$ first. Then you can run $getMinCut$ to obtain a Minimum Cut (vertices in the same part as src are returned).
Time: $\mathcal{O}\left(|V|^2|E|\right)$ for arbitrary networks. $\mathcal{O}\left(|E|\sqrt{|V|}\right)$ for bipartite/unit network. $\mathcal{O}\left(min|V|^{(}2/3), |E|^{(}1/2)|E|\right)$ 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 = f8bc377d2af3ac0d3b75bbacb2e4f7e9

  // 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 $G = (V, E)$.
Usage: Always run $mincostflow(src, sink)$ for some src and $sink$ to get the minimum cost and the maximum flow.
Time: $\mathcal{O}\left(|F||E|\log|E|\right)$ for non-negative costs. $\mathcal{O}\left(|V||E| + |F||E|\log|E|\right)$ 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 add(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 = c69ec434ecc34a1db966fd1b901850d2
```

link-cut.cpp
1 lines

```
// TODO
```

binary-lifting.cpp

Description: Compute the sparse table for binary lifting of a tree T.
Time: $\mathcal{O}\left(|V|\log|V|\right)$ for precalculation and $\mathcal{O}\left(\log|V|\right)$ for each lca query.
37 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 = 1c314be79fc6dee496617d2ec4f13616
```

cut-and-bridge.cpp

Description: Given an undirected graph $G = (V, E)$, compute all cut vertices and bridges.
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 edge i is in. $top[u]$ records the second highest vertex (which is unique) in the bcc which vertex u is in.
Time: $\mathcal{O}(|V| + |E|)$.

57 lines

```
struct VertexBCC {
  int n, bcc;
  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;
    }
  }
  bool SameBcc(int x, int y) {
    if (x == fa[top[y]] || y == fa[top[x]]) return 1;
    else return top[x] == top[y];
  }
  vector<pii> getBlockForest() { return bf; }
}; // hash-cpp-all = 909e9d5a16dbb2ec4031065b0eaabecd
```

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
```

dsu.cpp

Description: Disjoint set union. *merge* 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.

17 lines

```
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]); }

  // return 1 if x and y are in different component and
      ↪merge.
  bool merge(int x, int y) {
    int fx = getcomp(x), fy = getcomp(y);
    if (fx == fy) return 0;
```

```cpp
    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 $0, ..., N-1$. Use $top = top()$ to get a save point; use $undo(top)$ to go back to the save point.
Usage: Fill in struct T, function $join$ as well as choosing proper type (Z) for $glob$ and remember to initialize it. To undo, do in the following way:

```
Dsu dsu(n);
...
int top = dsu.top();
...  // do merging here.
dsu.undo(top);
```

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

54 lines

```cpp
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;
  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;

  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 = 4895f51f00e324e4caf81d76afe751f6
```

centroid-decomposition.cpp

Description: Centroid Decomposition.
Time: $\mathcal{O}(N \log N)$.

38 lines

```cpp
struct CentroidDecomposition {
  // anc[i]: ancestors of vertex i in centroid tree,
    ↪including itself.
  // dis[i]: distances from vertex i to ancestors of vertex
    ↪ i in centroid tree, not necessarily monotone.
  int n;
  vector<vi> anc, cdis;

  CentroidDecomposition(vector<vi> &g): n(sz(g)), anc(n),
    ↪cdis(n) {
    vi siz(n);
    vector<bool> vis(n);
    function<void(int, int)> solve = [&](int _, int tot) {
      int mn = inf, cent = -1;
      function<void(int, int)> getcent = [&](int now, int
        ↪fa) {
        siz[now] = 1;
        int mx = 0;
        for (auto v: g[now]) if (v != fa && vis[v] == 0) {
          getcent(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(_, -1); vis[cent] = 1;

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

      for (auto v: g[cent]) if (vis[v] == 0) solve(v, siz[v
        ↪] < siz[cent] ? siz[v] : tot - siz[cent]);
    };

    solve(0, n);
  }
}; // hash-cpp-all = 09f707d97935f6e7de36c112672c8214
```

heavy-light-decomposition.cpp

Description: Heavy Light Decomposition for a tree T (can be modified easily for forest).
Usage: g should be the adjacent list of the tree T. rt for specifying the root of the tree T (default 0). $chainApply(u, v, func, val)$ 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. $\mathcal{O}(\log N)$ for lca. $\mathcal{O}(\log |T| \cdot A)$ for chainApply / chainAsk, where A is the running time of $func$ in chainApply / chainAsk.

69 lines

```cpp
struct HLD {
  int n;
  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);
  }

  int lca(int u, int v) {
    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;
  }

  template<class... T>
  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.
  }
```

```cpp
  template<class T>
  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-all = fed861362ed14d707ccea2d6010bee89
```

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.

Usage: 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

```cpp
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
```

hopcroft.cpp

Description: Fast bipartite matching for bipartite graph. You can also get a vertex cover of a bipartite graph easily.

Time: $\mathcal{O}\left(|E|\sqrt{|V|}\right)$.

58 lines

```cpp
struct Hopcroft {
// hash-cpp-1
  int L, R;
  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

  // returns vertices matched to left part, -1 means not
    ↪matched.
  vi getMatch() { return 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
};
```

hungarian.cpp

Description: Given a complete bipartite graph $G = (L \cup, R, E)$, where $|L| \le |R|$, Finds minimum weighted perfect matching of L. Returns the matching.

Usage: $ws[i][j]$ 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

```cpp
template<class T = ll, T INF = numeric_limits<T>::max()>
vector<pii> Hungarian(const vector<vector<T>> &ws) {
  int L = sz(ws), R = 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 = 1247de71554b1d4764b16a36de08a191
```

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. 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 = 0;
    for (auto [x, y]: es) {
      g[x].push_back(m);
      if (!dir) g[y].push_back(m);
      m++;
    }

    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 = b7e06cbd0d08b9923de36919e27d67d8
```

String algorithms (5)

kmp.cpp

Description: Compute fail table of pattern string $s = s_0...s_{n-1}$ 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 $p_0...p_i$.
Usage: `KMP kmp(s)` for string s or `vector<int>` s.
Time: $\mathcal{O}(|p|)$ for precalculation and $\mathcal{O}(|p| + |t|)$ for matching. 26 lines

```
template<class T> struct KMP {
  const T s;
  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;
    }
  }

  // gets all matched (starting) positions.
  vi match(const T &t) {
    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-all = 35226020a90976c8bef2bc77416a917c
```

z-algo.cpp

Description: Given string $s = s_0...s_{n-1}$, compute array z where $z[i]$ is the lcp of $s_0...s_{n-1}$ and $s_i...s_{n-1}$. Use function $cal(t)$ (where $|t| = m$) to calculate the lcp of of $s_0...s_{n-1}$ and $t_i...t_{m-1}$ for each i.
Usage: `zAlgo za(s)` for string s or `vector<int>` s.
Time: $\mathcal{O}(|s|)$ for precalculation and $\mathcal{O}(|s| + |t|)$ for matching. 33 lines

```
template<class T>
struct zAlgo {
  const T s;
  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 - l], 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];
      }
    }
  }
  vi cal(const T &t) {
    int m = sz(t);
    vi res(m);
    int l = 0, r = 0;
    rep(i, 0, m - 1) {
      res[i] = max(0, min(i - l < n ? z[i - l] : 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-all = 0f63087b8b2527a427995e06cd7bb509
```

aho-corasick.cpp

Description: Aho Corasick Automaton of strings $s_0, ..., s_{n-1}$.
Usage: `AhoCorasick<'a', 26> ac;` for strings consisting of lowercase letters. Call $ac.build()$ after you insert all strings $s\_0, ..., s\_\{n-1\}$.
Time: $\mathcal{O}\left(\sum_{i=0}^{n-1} |s_i|\right)$. 47 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
```

suffix-array.cpp

Description: Suffix Array for non-cyclic string $s = s_0 ... s_{n-1}$. $rank[i]$ records the rank of the i-th suffix $s_i ... s_{n-1}$. $sa[i]$ 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.
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.
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: `Use SAM sam(s) for string` s `or vector<int>` s.
Time: $\mathcal{O}(|s|)$.

74 lines

```
template<class T> struct SAM {
  struct node { // hash-cpp-1
    map<int, int> nxt;
    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 = 1c12eb7fbeec418a5befc77214c19b9b

  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 = aadc726973415dfaac1e483d8fac558b
};
```

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 prefix tree. Note that fa[i] < i doesn't hold. occ should be set manually when building Trie T. root is 0.
Usage: `Use GSAM sam(T) for Trie` T, `where` T `is of type` $vector < GSAM :: node >$.
Time: $\mathcal{O}(|T|)$.

52 lines

```cpp
struct GSAM {
  struct node {
    map<int, int> nxt;
    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);
    }
  }
}; // hash-cpp-all = add4c78221df38584b76536f66703db7
```

manacher.cpp

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

12 lines

```cpp
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]] == 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 $s = s_0...s_{n-1}$, build the palindrom tree (automaton) for s. Each state of the automaton corresponds to a palindrome substring of s. Note that $t[i].fa < i$ holds.
Usage: `Palindrome pt(s)` for string s or `vector<int>` s.
Time: $\mathcal{O}(|s|)$.

36 lines

```cpp
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
```

hash-struct.cpp

Description: Hash struct. 1000000007 and 1000050131 are good moduli.

19 lines

```cpp
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{1ll * a.x * b.x, 1ll * 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
```

de-bruijin.cpp

1 lines

```cpp
// TODO
```

lyndon.cpp

1 lines

```cpp
// TODO
```

Math (6)

simplex.cpp

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \le b$, $x \ge 0$. 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 $res = 1$.
Time: $\mathcal{O}(NM * \#pivots)$, where N is the number of constraints and M is the number of variables.

71 lines

```cpp
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;

  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 = 65bffa3f1640fddb4ff878040d5c721c
```

berlekamp-massey.cpp

1 lines

```
// TODO
```

fft.cpp

Description: Fast Fourier Transform.
Time: $\mathcal{O}(N \log N)$

73 lines

```
// use T = double or long double.
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, 1ll * 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.
Usage: 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 is some commonly used *mod* and the corresponding primitive root.
$g \rightarrow mod$ $(\max \log(n))$
3 -> 104857601 (22), 167772161 (25), 469762049 (26), 998244353 (23), 1004535809 (21);
10 -> 786433 (18);
31 -> 2013265921 (27).
Time: $\mathcal{O}(N \log N)$.

50 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.

48 lines

```
template<class T> struct poly: vector<T> {
// hash-cpp-1
  using vector<T>::vector;
  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: 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 $a_0 + \ldots + a_{n-1}x^{n-1}$ and $Q(x)$, and then trimming to the proper degree.

Usage: Suppose $a\_i = \sum\_\{j=1\}\hat{}\{d\}a\_\{i-j\} * c\_j$, then just let $A =$ $a\_0, \ldots, a\_\{d-1$ and $C =$ $c\_1, \ldots, c\_d$.

"polynomial.cpp" 24 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
```

fast-subset-transform.cpp

Description: Fast Subtset Transform. Also known as fast zeta transform.

Usage: length of a should be a power of 2.

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

13 lines

```
template<class T> void fst(vector<T> &a) {
  int N = sz(a);
  for (int s = 1; s < N; s <<= 1) {
    rep(i, 0, N - 1) if (i & s) a[i] += a[i ^ s];
  }
}

template<class T> void ifst(vector<T> &a) {
  int N = sz(a);
  for (int s = 1; s < N; s <<= 1) {
    for (int i = N - 1; i >= 0; --i) if (i & s) a[i] -= a[i
      ^ s];
  }
} // hash-cpp-all = 1cc4c6746db79c729d29742ca3e210d1
```

fwht.cpp

Description: Fast Walsh-Hadamard Transform $fwt(a) = (\sum_i (-1)^{pc(i\&0)} a_i, \ldots, \sum_i (-1)^{pc(i\&n-1)} a_i)$. One can use it to do xor-convolution.

Usage: length of a should be a power of 2.

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

14 lines

```
template<class T> void fwt(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 = 39548d4e5eba54c67b841c6f77a928ed
```

fwht-eval.cpp

Description: Let $b = fwt(a)$. One can calculate b_{id} for some index id in $O(N)$ time.

Usage: length of a should be a power of 2.

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

9 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 = 70afad3ebf9c5d79cb34009e63ceab27
```

matroid.cpp

1 lines

```
// TODO
```

matrix.cpp

Description: Matrix struct. Used for Gaussian elimination or inverse of matrix.

Usage: To solve $Ax = b\hat{}\top$, call $SolveLinear(A, b)$. Besides, you need function $isZero$ for your template T.

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

98 lines

```
template<class T> struct Matrix {
  using Mat = Matrix;
  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]; }

  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;
  }

  // Do elimination for the first C columns, return the rank.
  int Gaussian(int C) {
    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;
  }

  Mat inverse() const {
    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;
  }

  friend pair<bool, Vec> SolveLinear(Mat A, const Vec &b) {
    #define revrep(i, a, n) for (auto i = n; i >= (a); --i)
    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())) 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])) {
        x -= A[i][j] * res[j];
        last = j;
      }
      if (last != -1) res[last] = x;
    }
    return {1, res};
  }
}; // hash-cpp-all = c32ead126cef68d15e8988daa6882258
```

linear-base.cpp

Description: Maximum weighted of Linear Base of vector space $\mathbb{Z}_2^{LG}$.
Usage: keep w[] zero to use unweighted Linear Base.
Time: $\mathcal{O}\left(LG \cdot \frac{LG}{w}\right)$ for insertion; $\mathcal{O}\left(LG^2 \cdot \frac{LG}{w}\right)$ for union.

56 lines

```
// T is the type of vectors and Z is the type of weights.
// w[i] is the non-negative weight of a[i].
template<int LG, class T = bitset<LG>, class Z = int> struct LB {
// hash-cpp-1
  #define revrep(i, a, n) for (auto i = n; i >= (a); --i)
  vector<T> a;
  vector<Z> w;

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

  // insert x. return 1 if the base is expanded.
  int insert(T x, Z val = 0) {
    revrep(i, 0, LG - 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 = a387f093648b516f28c7328018f56f16

  // min value 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, LG - 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 = 97b49d40578d7eb5b1beb46eb3348463

  // take the union of two bases.
  friend LB operator +(LB a, const LB &b) { // hash-cpp-3
    rep(i, 0, LG - 1) if (b[i] != 0) a.insert(b[i]);
    return a;
  } // hash-cpp-3 = 2cf1ecc88b178b24de182560d92f42d1

  // return the k-th smallest value spanned by vectors with wieght at least $val$. k starts from 0.
  // Time: O(LG \cdot \frac{LG}{w}).
  T kth(unsigned long long k, Z val = 0) { // hash-cpp-4
    int N = 0;
    rep(i, 0, LG - 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, LG - 1) if (a[i] != 0 && w[i] >= val) {
      --N;
      auto d = k >> N & 1;
      if (res[i] != d) res ^= a[i];
    }
    return res;
  } // hash-cpp-4 = 0d7e2a5d390ca813f8cfef6ac98d30d4
};
```

linear-base-intersect.cpp

Description: Intersection of two unweighted linear bases.
Usage: T should be of length at least $2 \cdot LG$.
Time: $\mathcal{O}\left(LG^2 \cdot \frac{LG}{w}\right)$.

15 lines

```
template<int LG, class T = bitset<LG * 2>> LB<LG, T> intersect(LB<LG, T> a, const LB<LG, T> &b) {
  LB<LG, T> res;
  rep(i, 0, LG - 1) if (a[i] != 0) a[i][LG + i] = 1;
  T msk(string(LG, '1'));
  rep(i, 0, LG - 1) {
    T x = a.ask_min(b[i]);
    if ((x & msk) != 0) a.insert(x);
    else {
      T y = 0;
      rep(j, 0, LG - 1) if (x[LG + j]) y ^= a[j];
      res.insert(y & msk);
    }
  }
  return res;
} // hash-cpp-all = ac77102be62217631c2b04f78b033fe2
```

Z3-vector.cpp

Description: vector in $\mathbb{Z}_3$.
Time: $\mathcal{O}(L/w)$.

38 lines

```
template<int L> struct v3 {
    bitset<L> a[3];
    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 operator +(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 operator *(rhs); }

    friend string to_string(const v3 &a) {
        string s;
        rep(i, 0, L - 1) s.push_back('0' + a[i]);
        return s;
    }
}; // hash-cpp-all = f7ad914469ba367fbd01711f4a2f1891
```

integrate.cpp

Description: Let $f(x)$ be a continuous function over $[a, b]$ having 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 $O(\frac{M(b-a)^5}{N^4})$.
Time: $\mathcal{O}(N \cdot T)$, where T is the time for evaluating f once.

8 lines

```
template<class T = db> T SimpsonsRule(const function<T(T)> &f, T a, T b, int N = 1'000) {
  T res = 0;
  T h = (b - a) / (N * 2);
  res += f(b);
  res += f(a);
  rep(i, 1, N * 2 - 1) res += f(a + h * i) * (i & 1 ? 4 : 2);
  return res * h / 3;
} // hash-cpp-all = 63c9ccf6ea860805cbbb606076a17671
```

integrate-adaptive.cpp

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

13 lines

```
template<class T = db> 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;
  };
  function<T(T, T, T, T, int)> rec = [&](T a, T b, T eps, T S, int dep) {
    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 rec(a, c, eps / 2, S1, dep - 1) + rec(c, b, eps / 2, S2, dep - 1);
  };
  return rec(a, b, eps, simpson(a, b), dep);
} // hash-cpp-all = 0a107d773979e044fd378bf28a451ed0
```

recursive-ternary-search.cpp

Description: for convex function $f : \mathbb{R}^d \to \mathbb{R}$, we can approximately find the global minimum using ternary search on each coordinate recursively.
Usage: 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.
Time: $\mathcal{O}\left(\log(1/\epsilon)^d \cdot T\right)$, where T is the time for evaluating the function f.

20 lines

```
// use T = double or long double.
template<class T> T rec_ters(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) {
      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 = 7463b827f8431abbabeed2f0528722ef
```

Number theory (7)

modnum.cpp

Description: Modular integer with $mod \le 2^{30} - 1$. 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 Mint1 / 3.pow(10) since 1 / 3 simply equals 0. Do not write things like Minta * b where a and b are int since you might first have integer overflow.
Usage: *mod* should be a global variable (either const int or int) and should satisfy $mod \le 2\hat{}\{30\} - 1$. for exmaple you can use like this:
const int mod = 998244353;
using Mint = Z<mod>;

32 lines

```
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 needed for ntt and polynomial operations.
// hash-cpp-2
  Z pow(ll k) const {
    Z res = 1, a = *this;
    for (; k; 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); } // just for debug.
};
```

factorization.cpp

Description: Fast Factorization. The mul function supports $0 \le a, b < c < 7.268 \times 10^{18}$ and is a little bit faster than __int128.
Time: $\mathcal{O}\left(n^{1/4}\right)$ for pollard-rho and same for the whole factorization.

63 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; 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 = fdf01d99eff9d68a0b5ba775f3086359

// hash-cpp-2
  mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
  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> work(ll n) {
    vector<ll> res;

    function<void(ll)> solve = [&](ll x) {
      if (x == 1) return;
      if (miller(x)) res.push_back(x);
      else {
        ll d = pollard(x);
        solve(d);
        solve(x / d);
      }
    };
    solve(n);
    return res;
  } // hash-cpp-2 = e51a9b9919035e8e774f8e4cff6b8a8a
}
```

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 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 = 1ll * 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
```