Team Reference Document

Heltion

September 30, 2023

Contents

1 Graph

1.1 Connected Components

1.1.1 Strongly Connected Components

Returns strongly connected components in topologically order.

```
vector<vector<int>>
strongly_connected_components(const vector<vector<int>> &g) {
  int n = g.size();
  vector<bool> done(n);
  vector<int> pos(n, -1), stack;
  vector<vector<int>> res;
  function<int(int)> dfs = [&](int u) {
    int low = pos[u] = stack.size();
    stack.push_back(u);
    for (int v : g[u]) {
      if (not done[v]) { low = min(low, ~pos[v] ? pos[v] : dfs(v)); }
    }
    if (low == pos[u]) {
      res.emplace_back(stack.begin() + low, stack.end());
      for (int v : res.back()) { done[v] = true; }
      stack.resize(low);
    }
    return low;
  };
  for (int i = 0; i < n; i += 1) {
    if (not done[i]) { dfs(i); }
  }
  ranges::reverse(res);
  return res;
}
```

1.1.2 Two-vertex-connected Components

```
vector<vector<int>>
two_vertex_connected_components(const vector<vector<int>> &g) {
  int n = g.size();
  vector<int> pos(n, -1), stack;
  vector<vector<int>> res;
  function<int(int, int)> dfs = [&](int u, int p) {
    int low = pos[u] = stack.size(), son = 0;
    stack.push_back(u);
    for (int v : g[u]) {
      if (v != p) {
        if (~pos[v]) {
          low = min(low, pos[v]);
        } else {
          int end = stack.size(), lowv = dfs(v, u);
          low = min(low, lowv);
          if (lowv >= pos[u] and (~p or son++)) {
            res.emplace_back(stack.begin() + end, stack.end());
            res.back().push_back(u);
            stack.resize(end);
          }
        }
      }
    }
    return low;
  };
  for (int i = 0; i < n; i += 1) {
    if (pos[i] == -1) {
      dfs(i, -1);
      res.emplace_back(move(stack));
    }
  }
  return res;
}
```

1.1.3 Two-edge-connected Components

```
vector<vector<int>> bcc(const vector<vector<int>> &g) {
  int n = g.size();
  vector<int> pos(n, -1), stack;
  vector<vector<int>> res;
  function<int(int, int)> dfs = [&](int u, int p) {
    int low = pos[u] = stack.size(), pc = 0;
    stack.push_back(u);
    for (int v : g[u]) {
      if (~pos[v]) {
        if (v != p or pc++) { low = min(low, pos[v]); }
      } else {
        low = min(low, dfs(v, u));
      }
    }
    if (low == pos[u]) {
      res.emplace_back(stack.begin() + low, stack.end());
      stack.resize(low);
    }
    return low;
  };
  for (int i = 0; i < n; i += 1) {
    if (pos[i] == -1) { dfs(i, -1); }
  }
  return res;
}
```

1.1.4 Three-edge-connected Components

```
vector<vector<int>>
three_edge_connected_components(const vector<vector<int>> &g) {
  int n = g.size(), dft = -1;
  vector<int> pre(n, -1), post(n), path(n, -1), low(n), deg(n);
  DisjointSetUnion dsu(n);
  function<void(int, int)> dfs = [&](int u, int p) {
    int pc = 0;
    low[u] = pre[u] = dft += 1;
    for (int v : g[u]) {
      if (v != u and (v != p or pc++)) {
        if (pre[v] != -1) {
          if (pre[v] < pre[u]) {
            deg[u] += 1;
            low[u] = min(low[u], pre[v]);
          } else {
            deg[u] -= 1;
            for (int &p = path[u];
                 p != -1 and pre[p] <= pre[v] and pre[v] <= post[p];) {
              dsu.merge(u, p);
```

```
              deg[u] += deg[p];
              p = path[p];
            }
          }
        } else {
          dfs(v, u);
          if (path[v] == -1 and deg[v] <= 1) {
            low[u] = min(low[u], low[v]);
            deg[u] += deg[v];
          } else {
            if (deg[v] == 0) { v = path[v]; }
            if (low[u] > low[v]) {
              low[u] = min(low[u], low[v]);
              swap(v, path[u]);
            }
            for (; v != -1; v = path[v]) {
              dsu.merge(u, v);
              deg[u] += deg[v];
            }
          }
        }
      }
    }
    post[u] = dft;
  };
  for (int i = 0; i < n; i += 1) {
    if (pre[i] == -1) { dfs(i, -1); }
  }
  vector<vector<int>> _res(n);
  for (int i = 0; i < n; i += 1) { _res[dsu.find(i)].push_back(i); }
  vector<vector<int>> res;
  for (auto &res_i : _res) {
    if (not res_i.empty()) { res.emplace_back(move(res_i)); }
  }
  return res;
}
```

1.2 Euler Walks

```
optional<vector<vector<pair<int, bool>>>>
undirected_walks(int n, const vector<pair<int, int>> &edges) {
  int m = ssize(edges);
  vector<vector<pair<int, bool>>> res;
  if (not m) { return res; }
  vector<vector<pair<int, bool>>> g(n);
  for (int i = 0; i < m; i += 1) {
    auto [u, v] = edges[i];
    g[u].emplace_back(i, true);
    g[v].emplace_back(i, false);
  }
  for (int i = 0; i < n; i += 1) {
    if (g[i].size() % 2) { return {}; }
  }
  vector<pair<int, bool>> walk;
  vector<bool> visited(m);
  vector<int> cur(n);
  function<void(int)> dfs = [&](int u) {
    for (int &i = cur[u]; i < ssize(g[u]);) {
      auto [j, d] = g[u][i];
      if (not visited[j]) {
        visited[j] = true;
        dfs(d ? edges[j].second : edges[j].first);
        walk.emplace_back(j, d);
      } else {
        i += 1;
      }
    }
  };
  for (int i = 0; i < n; i += 1) {
    dfs(i);
    if (not walk.empty()) {
      ranges::reverse(walk);
      res.emplace_back(move(walk));
    }
  }
  return res;
}
optional<vector<vector<int>>>
directed_walks(int n, const vector<pair<int, int>> &edges) {
  int m = ssize(edges);
  vector<vector<int>> res;
  if (not m) { return res; }
  vector<int> d(n);
  vector<vector<int>> g(n);
  for (int i = 0; i < m; i += 1) {
    auto [u, v] = edges[i];
    g[u].push_back(i);
    d[v] += 1;
  }
  for (int i = 0; i < n; i += 1) {
    if (ssize(g[i]) != d[i]) { return {}; }
  }
  vector<int> walk;
  vector<int> cur(n);
  vector<bool> visited(m);
  function<void(int)> dfs = [&](int u) {
    for (int &i = cur[u]; i < ssize(g[u]);) {
      int j = g[u][i];
      if (not visited[j]) {
        visited[j] = true;
        dfs(edges[j].second);
        walk.push_back(j);
      } else {
        i += 1;
      }
    }
  };
  for (int i = 0; i < n; i += 1) {
    dfs(i);
    if (not walk.empty()) {
      ranges::reverse(walk);
      res.emplace_back(move(walk));
    }
  }
  return res;
}
```

1.3 Dominator Tree

```
vector<int> dominator(const vector<vector<int>> &g, int s) {
  int n = g.size();
  vector<int> pos(n, -1), p, label(n), dom(n), sdom(n), dsu(n), par(n);
```

```
  vector<vector<int>> rg(n), bucket(n);
  function<void(int)> dfs = [&](int u) {
    int t = p.size();
    p.push_back(u);
    label[t] = sdom[t] = dsu[t] = pos[u] = t;
    for (int v : g[u]) {
      if (pos[v] == -1) {
        dfs(v);
        par[pos[v]] = t;
      }
      rg[pos[v]].push_back(t);
    }
  };
  function<int(int, int)> find = [&](int u, int x) {
    if (u == dsu[u]) { return x ? -1 : u; }
    int v = find(dsu[u], x + 1);
    if (v < 0) { return u; }
    if (sdom[label[dsu[u]]] < sdom[label[u]]) { label[u] = label[dsu[u]]; }
    dsu[u] = v;
    return x ? v : label[u];
  };
  dfs(s);
  iota(dom.begin(), dom.end(), 0);
  for (int i = ssize(p) - 1; i >= 0; i -= 1) {
    for (int j : rg[i]) { sdom[i] = min(sdom[i], sdom[find(j, 0)]); }
    if (i) { bucket[sdom[i]].push_back(i); }
    for (int k : bucket[i]) {
      int j = find(k, 0);
      dom[k] = sdom[j] == sdom[k] ? sdom[j] : j;
    }
    if (i > 1) { dsu[i] = par[i]; }
  }
  for (int i = 1; i < ssize(p); i += 1) {
    if (dom[i] != sdom[i]) { dom[i] = dom[dom[i]]; }
  }
  vector<int> res(n, -1);
  res[s] = s;
  for (int i = 1; i < ssize(p); i += 1) { res[p[i]] = p[dom[i]]; }
  return res;
}
```

1.4 Directed Minimum Spanning Tree

```
struct Node {
  Edge e;
  int d;
  Node *l, *r;
  Node(Edge e) : e(e), d(0) { l = r = nullptr; }
  void add(int v) {
    e.w += v;
    d += v;
  }
  void push() {
    if (l) { l->add(d); }
    if (r) { r->add(d); }
    d = 0;
  }
};
Node *merge(Node *u, Node *v) {
  if (not u or not v) { return u ?: v; }
  if (u->e.w > v->e.w) { swap(u, v); }
  u->push();
  u->r = merge(u->r, v);
  swap(u->l, u->r);
  return u;
}
void pop(Node *&u) {
  u->push();
  u = merge(u->l, u->r);
}
pair<i64, vector<int>>
directed_minimum_spanning_tree(int n, const vector<Edge> &edges, int s) {
  i64 ans = 0;
  vector<Node *> heap(n), edge(n);
  RollbackDisjointSetUnion dsu(n), rbdsu(n);
  vector<pair<Node *, int>> cycles;
  for (auto e : edges) { heap[e.v] = merge(heap[e.v], new Node(e)); }
  for (int i = 0; i < n; i += 1) {
    if (i == s) { continue; }
    for (int u = i;;) {
      if (not heap[u]) { return {}; }
      ans += (edge[u] = heap[u])->e.w;
      edge[u]->add(-edge[u]->e.w);
      int v = rbdsu.find(edge[u]->e.u);
      if (dsu.merge(u, v)) { break; }
      int t = rbdsu.time();
      while (rbdsu.merge(u, v)) {
        heap[rbdsu.find(u)] = merge(heap[u], heap[v]);
        u = rbdsu.find(u);
        v = rbdsu.find(edge[v]->e.u);
      }
      cycles.emplace_back(edge[u], t);
      while (heap[u] and rbdsu.find(heap[u]->e.u) == rbdsu.find(u)) {
        pop(heap[u]);
      }
    }
  }
  for (auto [p, t] : cycles | views::reverse) {
    int u = rbdsu.find(p->e.v);
    rbdsu.rollback(t);
    int v = rbdsu.find(edge[u]->e.v);
    edge[v] = exchange(edge[u], p);
  }
  vector<int> res(n, -1);
  for (int i = 0; i < n; i += 1) { res[i] = i == s ? i : edge[i]->e.u; }
  return {ans, res};
}
```

1.5 K Shortest Paths

```
struct Node {
  int v, h;
  i64 w;
  Node *l, *r;
  Node(int v, i64 w) : v(v), w(w), h(1) { l = r = nullptr; }
};
Node *merge(Node *u, Node *v) {
  if (not u or not v) { return u ?: v; }
  if (u->w > v->w) { swap(u, v); }
  Node *p = new Node(*u);
  p->r = merge(u->r, v);
  if (p->r and (not p->l or p->l->h < p->r->h)) { swap(p->l, p->r); }
```

```
  p->h = (p->r ? p->r->h : 0) + 1;
  return p;
}
struct Edge {
  int u, v, w;
};
template <typename T>
using minimum_heap = priority_queue<T, vector<T>, greater<T>>;
vector<i64> k_shortest_paths(int n, const vector<Edge> &edges, int s, int t,
                             int k) {
  vector<vector<int>> g(n);
  for (int i = 0; i < ssize(edges); i += 1) { g[edges[i].u].push_back(i); }
  vector<int> par(n, -1), p;
  vector<i64> d(n, -1);
  minimum_heap<pair<i64, int>> pq;
  pq.push({d[s] = 0, s});
  while (not pq.empty()) {
    auto [du, u] = pq.top();
    pq.pop();
    if (du > d[u]) { continue; }
    p.push_back(u);
    for (int i : g[u]) {
      auto [_, v, w] = edges[i];
      if (d[v] == -1 or d[v] > d[u] + w) {
        par[v] = i;
        pq.push({d[v] = d[u] + w, v});
      }
    }
  }
  if (d[t] == -1) { return vector<i64>(k, -1); }
  vector<Node *> heap(n);
  for (int i = 0; i < ssize(edges); i += 1) {
    auto [u, v, w] = edges[i];
    if (~d[u] and ~d[v] and par[v] != i) {
      heap[v] = merge(heap[v], new Node(u, d[u] + w - d[v]));
    }
  }
  for (int u : p) {
    if (u != s) { heap[u] = merge(heap[u], heap[edges[par[u]].u]); }
  }
  minimum_heap<pair<i64, Node *>> q;
  if (heap[t]) { q.push({d[t] + heap[t]->w, heap[t]}); }
  vector<i64> res = {d[t]};
  for (int i = 1; i < k and not q.empty(); i += 1) {
    auto [w, p] = q.top();
    q.pop();
    res.push_back(w);
    if (heap[p->v]) { q.push({w + heap[p->v]->w, heap[p->v]}); }
    for (auto c : {p->l, p->r}) {
      if (c) { q.push({w + c->w - p->w, c}); }
    }
  }
  res.resize(k, -1);
  return res;
}
```

1.6 Global Minimum Cut

```
i64 global_minimum_cut(vector<vector<i64>> &w) {
  int n = w.size();
  if (n == 2) { return w[0][1]; }
  vector<bool> in(n);
  vector<int> add;
  vector<i64> s(n);
  i64 st = 0;
  for (int i = 0; i < n; i += 1) {
    int k = -1;
    for (int j = 0; j < n; j += 1) {
      if (not in[j]) {
        if (k == -1 or s[j] > s[k]) { k = j; }
      }
    }
    add.push_back(k);
    st = s[k];
    in[k] = true;
    for (int j = 0; j < n; j += 1) { s[j] += w[j][k]; }
  }
  for (int i = 0; i < n; i += 1) {}
  int x = add.rbegin()[1], y = add.back();
  if (x == n - 1) { swap(x, y); }
  for (int i = 0; i < n; i += 1) {
    swap(w[y][i], w[n - 1][i]);
    swap(w[i][y], w[i][n - 1]);
  }
  for (int i = 0; i + 1 < n; i += 1) {
    w[i][x] += w[i][n - 1];
    w[x][i] += w[n - 1][i];
  }
  w.pop_back();
  return min(st, stoer_wagner(w));
}
```

1.7 Minimum Perfect Matching on Bipartite Graph

```
minimum_perfect_matching_on_bipartite_graph(const vector<vector<i64>> &w) {
  i64 n = w.size();
  vector<int> rm(n, -1), cm(n, -1);
  vector<i64> pi(n);
  auto resid = [&](int r, int c) { return w[r][c] - pi[c]; };
  for (int c = 0; c < n; c += 1) {
    int r = ranges::min(views::iota(0, n), {}, [&](int r) { return w[r][c]; });
    pi[c] = w[r][c];
    if (rm[r] == -1) {
      rm[r] = c;
      cm[c] = r;
    }
  }
  vector<int> cols(n);
  iota(cols.begin(), cols.end(), 0);
  for (int r = 0; r < n; r += 1) {
    if (rm[r] != -1) { continue; }
    vector<i64> d(n);
    for (int c = 0; c < n; c += 1) { d[c] = resid(r, c); }
    vector<int> pre(n, r);
    int scan = 0, label = 0, last = 0, col = -1;
    [&]() {
      while (true) {
        if (scan == label) {
          last = scan;
          i64 min = d[cols[scan]];
          for (int j = scan; j < n; j += 1) {
            int c = cols[j];
```

```cpp
            if (d[c] <= min) {
              if (d[c] < min) {
                min = d[c];
                label = scan;
              }
              swap(cols[j], cols[label++]);
            }
          }
          for (int j = scan; j < label; j += 1) {
            if (int c = cols[j]; cm[c] == -1) {
              col = c;
              return;
            }
          }
        }
        int c1 = cols[scan++], r1 = cm[c1];
        for (int j = label; j < n; j += 1) {
          int c2 = cols[j];
          i64 len = resid(r1, c2) - resid(r1, c1);
          if (d[c2] > d[c1] + len) {
            d[c2] = d[c1] + len;
            pre[c2] = r1;
            if (len == 0) {
              if (cm[c2] == -1) {
                col = c2;
                return;
              }
              swap(cols[j], cols[label++]);
            }
          }
        }
      }
    }();
    for (int i = 0; i < last; i += 1) {
      int c = cols[i];
      pi[c] += d[c] - d[col];
    }
    for (int t = col; t != -1;) {
      col = t;
      int r = pre[col];
      cm[col] = r;
      swap(rm[r], t);
    }
  }
  i64 res = 0;
  for (int i = 0; i < n; i += 1) { res += w[i][rm[i]]; }
  return {res, rm};
}
```

1.8 Matching on General Graph

```cpp
vector<int> matching(const vector<vector<int>> &g) {
  int n = g.size();
  int mark = 0;
  vector<int> matched(n, -1), par(n, -1), book(n);
  auto match = [&](int s) {
    vector<int> c(n), type(n, -1);
    iota(c.begin(), c.end(), 0);
    queue<int> q;
    q.push(s);
    type[s] = 0;
    while (not q.empty()) {
      int u = q.front();
      q.pop();
      for (int v : g[u])
        if (type[v] == -1) {
          par[v] = u;
          type[v] = 1;
          int w = matched[v];
          if (w == -1) {
            [&](int u) {
              while (u != -1) {
                int v = matched[par[u]];
                matched[matched[u] = par[u]] = u;
                u = v;
              }
            }(v);
            return;
          }
          q.push(w);
          type[w] = 0;
        } else if (not type[v] and c[u] != c[v]) {
          int w = [&](int u, int v) {
            mark += 1;
            while (true) {
              if (u != -1) {
                if (book[u] == mark) { return u; }
                book[u] = mark;
                u = c[par[matched[u]]];
              }
              swap(u, v);
            }
          }(u, v);
          auto up = [&](int u, int v, int w) {
            while (c[u] != w) {
              par[u] = v;
              v = matched[u];
              if (type[v] == 1) {
                q.push(v);
                type[v] == 0;
              }
              if (c[u] == u) { c[u] = w; }
              if (c[v] == v) { c[v] = w; }
              u = par[v];
            }
          };
          up(u, v, w);
          up(v, u, w);
          for (int i = 0; i < n; i += 1) { c[i] = c[c[i]]; }
        }
    }
  };
  for (int i = 0; i < n; i += 1) {
    if (matched[i] == -1) { match(i); }
  }
  return matched;
}
```

1.9 Maximum Flow

```cpp
struct HighestLabelPreflowPush {
  int n;
```

```
  vector<vector<int>> g;
  vector<Edge> edges;
  HighestLabelPreflowPush(int n) : n(n), g(n) {}
  int add(int u, int v, i64 f) {
    if (u == v) { return -1; }
    int i = ssize(edges);
    edges.push_back({u, v, f});
    g[u].push_back(i);
    edges.push_back({v, u, 0});
    g[v].push_back(i + 1);
    return i;
  }
  i64 max_flow(int s, int t) {
    vector<i64> p(n);
    vector<int> h(n), cur(n), count(n * 2);
    vector<vector<int>> pq(n * 2);
    auto push = [&](int i, i64 f) {
      auto [u, v, _] = edges[i];
      if (not p[v] and f) { pq[h[v]].push_back(v); }
      edges[i].f -= f;
      edges[i ^ 1].f += f;
      p[u] -= f;
      p[v] += f;
    };
    h[s] = n;
    count[0] = n - 1;
    p[t] = 1;
    for (int i : g[s]) { push(i, edges[i].f); }
    for (int hi = 0;;) {
      while (pq[hi].empty()) {
        if (not hi--) { return -p[s]; }
      }
      int u = pq[hi].back();
      pq[hi].pop_back();
      while (p[u] > 0) {
        if (cur[u] == ssize(g[u])) {
          h[u] = n * 2 + 1;
          for (int i = 0; i < ssize(g[u]); i += 1) {
            auto [_, v, f] = edges[g[u][i]];
            if (f and h[u] > h[v] + 1) {
              h[u] = h[v] + 1;
              cur[u] = i;
            }
          }
          count[h[u]] += 1;
          if (not(count[hi] -= 1) and hi < n) {
            for (int i = 0; i < n; i += 1) {
              if (h[i] > hi and h[i] < n) {
                count[h[i]] -= 1;
                h[i] = n + 1;
              }
            }
          }
          hi = h[u];
        } else {
          int i = g[u][cur[u]];
          auto [_, v, f] = edges[i];
          if (f and h[u] == h[v] + 1) {
            push(i, min(p[u], f));
          } else {
            cur[u] += 1;
          }
        }
      }
    }
    return i64(0);
  }
};
```

1.10 Minimum Cost Maximum Flow

Constraints: there is no edge with negative cost.

```
struct MinimumCostMaximumFlow {
  template <typename T>
  using minimum_heap = priority_queue<T, vector<T>, greater<T>>;
  int n;
  vector<Edge> edges;
  vector<vector<int>> g;
  MinimumCostMaximumFlow(int n) : n(n), g(n) {}
  int add_edge(int u, int v, i64 f, i64 c) {
    int i = edges.size();
    edges.push_back({u, v, f, c});
    edges.push_back({v, u, 0, -c});
    g[u].push_back(i);
    g[v].push_back(i + 1);
    return i;
  }
  pair<i64, i64> flow(int s, int t) {
    constexpr i64 inf = numeric_limits<i64>::max();
    vector<i64> d, h(n);
    vector<int> p;
    auto dijkstra = [&]() {
      d.assign(n, inf);
      p.assign(n, -1);
      minimum_heap<pair<i64, int>> q;
      q.emplace(d[s] = 0, s);
      while (not q.empty()) {
        auto [du, u] = q.top();
        q.pop();
        if (du > d[u]) { continue; }
        for (int i : g[u]) {
          auto [_, v, f, c] = edges[i];
          if (f and d[v] > d[u] + h[u] - h[v] + c) {
            p[v] = i;
            q.emplace(d[v] = d[u] + h[u] - h[v] + c, v);
          }
        }
      }
      return ~p[t];
    };
    i64 f = 0, c = 0;
    while (dijkstra()) {
      for (int i = 0; i < n; i += 1) { h[i] += d[i]; }
      vector<int> path;
      for (int u = t; u != s; u = edges[p[u]].u) { path.push_back(p[u]); }
      i64 mf =
          edges[ranges::min(path, {}, [&](int i) { return edges[i].f; })].f;
      f += mf;
      c += mf * h[t];
      for (int i : path) {
        edges[i].f -= mf;
        edges[i ^ 1].f += mf;
      }
    }
```

```
    return {f, c};
  }
};
```

2 Data Structure

2.1 Disjoint Set Union

```
struct DisjointSetUnion {
  vector<int> dsu;
  DisjointSetUnion(int n) : dsu(n, -1) {}
  int find(int u) { return dsu[u] < 0 ? u : dsu[u] = find(dsu[u]); }
  void merge(int u, int v) {
    u = find(u);
    v = find(v);
    if (u != v) {
      if (dsu[u] > dsu[v]) { swap(u, v); }
      dsu[u] += dsu[v];
      dsu[v] = u;
    }
  }
};
struct RollbackDisjointSetUnion {
  vector<pair<int, int>> stack;
  vector<int> dsu;
  RollbackDisjointSetUnion(int n) : dsu(n, -1) {}
  int find(int u) { return dsu[u] < 0 ? u : find(dsu[u]); }
  int time() { return ssize(stack); }
  bool merge(int u, int v) {
    if ((u = find(u)) == (v = find(v))) { return false; }
    if (dsu[u] < dsu[v]) { swap(u, v); }
    stack.emplace_back(u, dsu[u]);
    dsu[v] += dsu[u];
    dsu[u] = v;
    return true;
  }
  void rollback(int t) {
    while (ssize(stack) > t) {
      auto [u, dsu_u] = stack.back();
      stack.pop_back();
      dsu[dsu[u]] -= dsu_u;
      dsu[u] = dsu_u;
    }
  }
};
```

2.2 Sparse Table

```
struct SparseTable {
  vector<vector<int>> table;
  SparseTable() {}
  SparseTable(const vector<int> &a) {
    int n = a.size(), h = bit_width(a.size());
    table.resize(h);
    table[0] = a;
    for (int i = 1; i < h; i += 1) {
      table[i].resize(n - (1 << i) + 1);
      for (int j = 0; j + (1 << i) <= n; j += 1) {
        table[i][j] = min(table[i - 1][j], table[i - 1][j + (1 << (i - 1))]);
      }
    }
  }
  int query(int l, int r) {
    int h = bit_width(unsigned(r - l)) - 1;
    return min(table[h][l], table[h][r - (1 << h)]);
  }
};
struct DisjointSparseTable {
  vector<vector<int>> table;
  DisjointSparseTable(const vector<int> &a) {
    int h = bit_width(a.size() - 1), n = a.size();
    table.resize(h, a);
    for (int i = 0; i < h; i += 1) {
      for (int j = 0; j + (1 << i) < n; j += (2 << i)) {
        for (int k = j + (1 << i) - 2; k >= j; k -= 1) {
          table[i][k] = min(table[i][k], table[i][k + 1]);
        }
        for (int k = j + (1 << i) + 1; k < j + (2 << i) and k < n; k += 1) {
          table[i][k] = min(table[i][k], table[i][k - 1]);
        }
      }
    }
  }
  int query(int l, int r) {
    if (l + 1 == r) { return table[0][l]; }
    int i = bit_width(unsigned(l ^ (r - 1))) - 1;
    return min(table[i][l], table[i][r - 1]);
  }
};
```

2.3 Treap

```
struct Node {
  static constexpr bool persistent = true;
  static mt19937_64 mt;
  Node *l, *r;
  u64 priority;
  int size, v;
  i64 sum;
  Node(const Node &other) { memcpy(this, &other, sizeof(Node)); }
  Node(int v) : v(v), sum(v), priority(mt()), size(1) { l = r = nullptr; }
  Node *update(Node *l, Node *r) {
    Node *p = persistent ? new Node(*this) : this;
    p->l = l;
    p->r = r;
    p->size = (l ? l->size : 0) + 1 + (r ? r->size : 0);
    p->sum = (l ? l->sum : 0) + v + (r ? r->sum : 0);
    return p;
  }
};
mt19937_64 Node::mt;
pair<Node *, Node *> split_by_v(Node *p, int v) {
  if (not p) { return {}; }
  if (p->v < v) {
    auto [l, r] = split_by_v(p->r, v);
    return {p->update(p->l, l), r};
  }
  auto [l, r] = split_by_v(p->l, v);
```

```
  return {l, p->update(r, p->r)};
}
pair<Node *, Node *> split_by_size(Node *p, int size) {
  if (not p) { return {}; }
  int l_size = p->l ? p->l->size : 0;
  if (l_size < size) {
    auto [l, r] = split_by_size(p->r, size - l_size - 1);
    return {p->update(p->l, l), r};
  }
  auto [l, r] = split_by_size(p->l, size);
  return {l, p->update(r, p->r)};
}
Node *merge(Node *l, Node *r) {
  if (not l or not r) { return l ?: r; }
  if (l->priority < r->priority) { return r->update(merge(l, r->l), r->r); }
  return l->update(l->l, merge(l->r, r));
}
```

2.4 Lines Maximum

```
struct Line {
  static bool q;
  mutable i64 k, b, p;
  bool operator<(const Line &rhs) const { return q ? p < rhs.p : k < rhs.k; }
};
bool Line::q = false;
struct Lines : multiset<Line> {
  static constexpr i64 inf = numeric_limits<i64>::max();
  static i64 div(i64 a, i64 b) { return a / b - ((a ^ b) < 0 and a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) { return x->p = inf, false; }
    if (x->k == y->k) {
      x->p = x->b > y->b ? inf : -inf;
    } else {
      x->p = div(y->b - x->b, x->k - y->k);
    }
    return x->p >= y->p;
  }
  void add(i64 k, i64 b) {
    Line::q = false;
    auto z = insert({k, b, 0}), y = z++, x = y;
    while (isect(y, z)) { z = erase(z); }
    if (x != begin() and isect(--x, y)) { isect(x, y = erase(y)); }
    while ((y = x) != begin() and (--x)->p >= y->p) { isect(x, erase(y)); }
  }
  optional<i64> get(i64 x) {
    if (empty()) { return {}; }
    Line::q = true;
    auto it = lower_bound({0, 0, x});
    return it->k * x + it->b;
  }
};
```

2.5 Segments Maximum

```
struct Segment {
  i64 k, b;
  i64 get(i64 x) { return k * x + b; }
};
struct Segments {
  struct Node {
    optional&lt;Segment&gt; s;
    Node *l, *r;
  };
  i64 tl, tr;
  Node *root;
  Segments(i64 tl, i64 tr) : tl(tl), tr(tr), root(nullptr) {}
  void add(i64 l, i64 r, i64 k, i64 b) {
    function<void(Node *&, i64, i64, Segment)> rec = [&](Node *&p, i64 tl,
                                                          i64 tr, Segment s) {
      if (p == nullptr) { p = new Node(); }
      i64 tm = midpoint(tl, tr);
      if (tl >= l and tr <= r) {
        if (not p->s) {
          p->s = s;
          return;
        }
        auto t = p->s.value();
        if (t.get(tl) >= s.get(tl)) {
          if (t.get(tr) >= s.get(tr)) { return; }
          if (t.get(tm) >= s.get(tm)) { return rec(p->r, tm + 1, tr, s); }
          p->s = s;
          return rec(p->l, tl, tm, t);
        }
        if (t.get(tr) <= s.get(tr)) {
          p->s = s;
          return;
        }
        if (t.get(tm) <= s.get(tm)) {
          p->s = s;
          return rec(p->r, tm + 1, tr, t);
        }
        return rec(p->l, tl, tm, s);
      }
      if (l <= tm) { rec(p->l, tl, tm, s); }
      if (r > tm) { rec(p->r, tm + 1, tr, s); }
    };
    rec(root, tl, tr, {k, b});
  }
  optional<i64> get(i64 x) {
    optional<i64> res = {};
    function<void(Node *, i64, i64)> rec = [&](Node *p, i64 tl, i64 tr) {
      if (p == nullptr) { return; }
      i64 tm = midpoint(tl, tr);
      if (p->s) {
        i64 y = p->s.value().get(x);
        if (not res or res.value() < y) { res = y; }
      }
      if (x <= tm) {
        rec(p->l, tl, tm);
      } else {
        rec(p->r, tm + 1, tr);
      }
    };
    rec(root, tl, tr);
    return res;
  }
};
```

2.6 Segment Beats

```cpp
struct Mv {
  static constexpr i64 inf = numeric_limits<i64>::max() / 2;
  i64 mv, smv, cmv, tmv;
  bool less;
  i64 def() { return less ? inf : -inf; }
  i64 mmv(i64 x, i64 y) { return less ? min(x, y) : max(x, y); }
  Mv(i64 x, bool less) : less(less) {
    mv = x;
    smv = tmv = def();
    cmv = 1;
  }
  void up(const Mv &ls, const Mv &rs) {
    mv = mmv(ls.mv, rs.mv);
    smv = mmv(ls.mv == mv ? ls.smv : ls.mv, rs.mv == mv ? rs.smv : rs.mv);
    cmv = (ls.mv == mv ? ls.cmv : 0) + (rs.mv == mv ? rs.cmv : 0);
  }
  void add(i64 x) {
    mv += x;
    if (smv != def()) { smv += x; }
    if (tmv != def()) { tmv += x; }
  }
};
struct Node {
  Mv mn, mx;
  i64 sum, tsum;
  Node *ls, *rs;
  Node(i64 x = 0) : sum(x), tsum(0), mn(x, true), mx(x, false) {
    ls = rs = nullptr;
  }
  void up() {
    sum = ls->sum + rs->sum;
    mx.up(ls->mx, rs->mx);
    mn.up(ls->mn, rs->mn);
  }
  void down(int tl, int tr) {
    if (tsum) {
      int tm = midpoint(tl, tr);
      ls->add(tl, tm, tsum);
      rs->add(tm, tr, tsum);
      tsum = 0;
    }
    if (mn.tmv != mn.def()) {
      ls->ch(mn.tmv, true);
      rs->ch(mn.tmv, true);
      mn.tmv = mn.def();
    }
    if (mx.tmv != mx.def()) {
      ls->ch(mx.tmv, false);
      rs->ch(mx.tmv, false);
      mx.tmv = mx.def();
    }
  }
  bool cmp(i64 x, i64 y, bool less) { return less ? x < y : x > y; }
  void add(int tl, int tr, i64 x) {
    sum += (tr - tl) * x;
    tsum += x;
    mx.add(x);
    mn.add(x);
  }
  void ch(i64 x, bool less) {
    auto &lhs = less ? mn : mx, &rhs = less ? mx : mn;
    if (not cmp(x, rhs.mv, less)) { return; }
    sum += (x - rhs.mv) * rhs.cmv;
    if (lhs.smv == rhs.mv) { lhs.smv = x; }
    if (lhs.mv == rhs.mv) { lhs.mv = x; }
    if (cmp(x, rhs.tmv, less)) { rhs.tmv = x; }
    rhs.mv = lhs.tmv = x;
  }
  void add(int tl, int tr, int l, int r, i64 x) {
    if (tl >= l and tr <= r) { return add(tl, tr, x); }
    down(tl, tr);
    int tm = midpoint(tl, tr);
    if (l < tm) { ls->add(tl, tm, l, r, x); }
    if (r > tm) { rs->add(tm, tr, l, r, x); }
    up();
  }
  void ch(int tl, int tr, int l, int r, i64 x, bool less) {
    auto &lhs = less ? mn : mx, &rhs = less ? mx : mn;
    if (not cmp(x, rhs.mv, less)) { return; }
    if (tl >= l and tr <= r and cmp(rhs.smv, x, less)) { return ch(x, less); }
    down(tl, tr);
    int tm = midpoint(tl, tr);
    if (l < tm) { ls->ch(tl, tm, l, r, x, less); }
    if (r > tm) { rs->ch(tm, tr, l, r, x, less); }
    up();
  }
  i64 get(int tl, int tr, int l, int r) {
    if (tl >= l and tr <= r) { return sum; }
    down(tl, tr);
    i64 res = 0;
    int tm = midpoint(tl, tr);
    if (l < tm) { res += ls->get(tl, tm, l, r); }
    if (r > tm) { res += rs->get(tm, tr, l, r); }
    return res;
  }
};
```

2.7 Tree

2.7.1 Least Common Ancestor

```cpp
struct LeastCommonAncestor {
  SparseTable st;
  vector<int> p, time, a, par;
  LeastCommonAncestor(int root, const vector<vector<int>> &g) {
    int n = g.size();
    time.resize(n, -1);
    par.resize(n, -1);
    function<void(int)> dfs = [&](int u) {
      time[u] = p.size();
      p.push_back(u);
      for (int v : g[u]) {
        if (time[v] == -1) {
          par[v] = u;
          dfs(v);
        }
      }
    };
    dfs(root);
    a.resize(n);
    for (int i = 1; i < n; i += 1) { a[i] = time[par[p[i]]]; }
    st = SparseTable(a);
```

```
  }
  int query(int u, int v) {
    if (u == v) { return u; }
    if (time[u] > time[v]) { swap(u, v); }
    return p[st.query(time[u] + 1, time[v] + 1)];
  }
};
```

2.7.2 Link Cut Tree

```
struct Node {
  i64 v, sum;
  array<Node *, 2> c;
  Node *p;
  bool flip;
  Node(i64 v) : v(v), sum(v), p(nullptr) { c.fill(nullptr); }
  int side() {
    if (not p) { return -1; }
    if (p->c[0] == this) { return 0; }
    if (p->c[1] == this) { return 1; }
    return -1;
  }
  void up() { sum = (c[0] ? c[0]->sum : 0) + v + (c[1] ? c[1]->sum : 0); }
  void down() {
    if (flip) {
      swap(c[0], c[1]);
      if (c[0]) { c[0]->flip ^= 1; }
      if (c[1]) { c[1]->flip ^= 1; }
      flip ^= 1;
    }
  }
  void attach(int s, Node *u) {
    c[s] = u;
    if (u) { u->p = this; }
    up();
  }
  void rotate() {
    auto p = this->p;
    auto pp = p->p;
    int s = side();
    int ps = p->side();
    auto b = c[s ^ 1];
    p->attach(s, b);
    attach(s ^ 1, p);
    if (~ps) { pp->attach(ps, this); }
    this->p = pp;
  }
  void splay() {
    down();
    while (side() >= 0 and p->side() >= 0) {
      p->p->down();
      p->down();
      down();
      (side() == p->side() ? p : this)->rotate();
      rotate();
    }
    if (side() >= 0) {
      p->down();
      down();
      rotate();
    }
  }
  void access() {
    splay();
    attach(1, nullptr);
    while (p != nullptr) {
      auto w = p;
      w->splay();
      w->attach(1, this);
      rotate();
    }
  }
  void reroot() {
    access();
    flip ^= 1;
    down();
  }
  void link(Node *u) {
    u->reroot();
    access();
    attach(1, u);
  }
  void cut(Node *u) {
    u->reroot();
    access();
    if (c[0] == u) {
      c[0] = nullptr;
      u->p = nullptr;
      up();
    }
  }
};
```

3 String

3.1 Z

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

3.2 Lyndon Factorization

```
vector<int> lyndon_factorization(string const &s) {
  vector<int> res = {0};
  for (int i = 0, n = s.size(); i < n;) {
    int j = i + 1, k = i;
    for (; j < n and s[k] <= s[j]; j += 1) { k = s[k] < s[j] ? i : k + 1; }
    while (i <= k) { res.push_back(i += j - k); }
  }
  return res;
```

```
}
```

3.3 Border

```
vector<int> fborder(const string &s) {
  int n = s.size();
  vector<int> res(n);
  for (int i = 1; i < n; i += 1) {
    int &j = res[i] = res[i - 1];
    while (j and s[i] != s[j]) { j = res[j - 1]; }
    j += s[i] == s[j];
  }
  return res;
}
```

3.4 Manacher

```
vector<int> manacher(const string &s) {
  int n = s.size();
  vector<int> p(n);
  for (int i = 0, j = 0; i < n; i += 1) {
    if (j + p[j] > i) { p[i] = min(p[j * 2 - i], j + p[j] - i); }
    while (i >= p[i] and i + p[i] < n and s[i - p[i]] == s[i + p[i]]) {
      p[i] += 1;
    }
    if (i + p[i] > j + p[j]) { j = i; }
  }
  return p;
}
```

3.5 Suffix Array

```
pair<vector<int>, vector<int>> binary_lifting(const string &s) {
  int n = s.size(), k = 0;
  vector<int> p(n), rank(n), q, count;
  iota(p.begin(), p.end(), 0);
  ranges::sort(p, {}, [&](int i) { return s[i]; });
  for (int i = 0; i < n; i += 1) {
    rank[p[i]] = i and s[p[i]] == s[p[i - 1]] ? rank[p[i - 1]] : k++;
  }
  for (int m = 1; m < n; m *= 2) {
    q.resize(m);
    iota(q.begin(), q.end(), n - m);
    for (int i : p) {
      if (i >= m) { q.push_back(i - m); }
    }
    count.assign(k, 0);
    for (int i : rank) { count[i] += 1; }
    partial_sum(count.begin(), count.end(), count.begin());
    for (int i = n - 1; i >= 0; i -= 1) { p[count[rank[q[i]]] -= 1] = q[i]; }
    auto previous = rank;
    previous.resize(2 * n, -1);
    k = 0;
    for (int i = 0; i < n; i += 1) {
      rank[p[i]] = i and previous[p[i]] == previous[p[i - 1]] and
                       previous[p[i] + m] == previous[p[i - 1] + m]
                   ? rank[p[i - 1]]
                   : k++;
    }
  }
  vector<int> lcp(n);
  k = 0;
  for (int i = 0; i < n; i += 1) {
    if (rank[i]) {
      k = max(k - 1, 0);
      int j = p[rank[i] - 1];
      while (i + k < n and j + k < n and s[i + k] == s[j + k]) { k += 1; }
      lcp[rank[i]] = k;
    }
  }
  return {p, lcp};
}
```

3.6 Aho-Corasick Automaton

```
constexpr int sigma = 26;
struct Node {
  int link;
  array<int, sigma> next;
  Node() : link(0) { next.fill(0); }
};
struct AhoCorasick : vector<Node> {
  AhoCorasick() : vector<Node>(1) {}
  int add(const string &s, char first = 'a') {
    int p = 0;
    for (char si : s) {
      int c = si - first;
      if (not at(p).next[c]) {
        at(p).next[c] = size();
        emplace_back();
      }
      p = at(p).next[c];
    }
    return p;
  }
  void init() {
    queue<int> q;
    for (int i = 0; i < sigma; i += 1) {
      if (at(0).next[i]) { q.push(at(0).next[i]); }
    }
    while (not q.empty()) {
      int u = q.front();
      q.pop();
      for (int i = 0; i < sigma; i += 1) {
        if (at(u).next[i]) {
          at(at(u).next[i]).link = at(at(u).link).next[i];
          q.push(at(u).next[i]);
        } else {
          at(u).next[i] = at(at(u).link).next[i];
        }
      }
    }
  }
};
```

3.7 Suffix Automaton

```
struct Node {
  int link, len;
  array<int, sigma> next;
  Node() : link(-1), len(0) { next.fill(-1); }
};
struct SuffixAutomaton : vector<Node> {
  SuffixAutomaton() : vector<Node>(1) {}
  int extend(int p, int c) {
    if (~at(p).next[c]) {
      // For online multiple strings.
      int q = at(p).next[c];
      if (at(p).len + 1 == at(q).len) { return q; }
      int clone = size();
      push_back(at(q));
      back().len = at(p).len + 1;
      while (~p and at(p).next[c] == q) {
        at(p).next[c] = clone;
        p = at(p).link;
      }
      at(q).link = clone;
      return clone;
    }
    int cur = size();
    emplace_back();
    back().len = at(p).len + 1;
    while (~p and at(p).next[c] == -1) {
      at(p).next[c] = cur;
      p = at(p).link;
    }
    if (~p) {
      int q = at(p).next[c];
      if (at(p).len + 1 == at(q).len) {
        back().link = q;
      } else {
        int clone = size();
        push_back(at(q));
        back().len = at(p).len + 1;
        while (~p and at(p).next[c] == q) {
          at(p).next[c] = clone;
          p = at(p).link;
        }
        at(q).link = at(cur).link = clone;
      }
    } else {
      back().link = 0;
    }
    return cur;
  }
};
```

3.8 Palindromic Tree

```
struct Node {
  int sum, len, link;
  array<int, sigma> next;
  Node(int len) : len(len) {
    sum = link = 0;
    next.fill(0);
  }
};
struct PalindromicTree : vector<Node> {
  int last;
  vector<int> s;
  PalindromicTree() : last(0) {
    emplace_back(0);
    emplace_back(-1);
    at(0).link = 1;
  }
  int get_link(int u, int i) {
    while (i < at(u).len + 1 or s[i - at(u).len - 1] != s[i]) u = at(u).link;
    return u;
  }
  void extend(int i) {
    int cur = get_link(last, i);
    if (not at(cur).next[s[i]]) {
      int now = size();
      emplace_back(at(cur).len + 2);
      back().link = at(get_link(at(cur).link, i)).next[s[i]];
      back().sum = at(back().link).sum + 1;
      at(cur).next[s[i]] = now;
    }
    last = at(cur).next[s[i]];
  }
};
```

4 Number Theory

4.1 Modular Arithmetic

4.1.1 Sqrt

Find x such that $x^2 \equiv y \pmod{p}$.
Constraints: p is prime and $0 \leq y < p$.

```
i64 sqrt(i64 y, i64 p) {
  static mt19937_64 mt;
  if (y <= 1) { return y; };
  if (power(y, (p - 1) / 2, p) != 1) { return -1; }
  uniform_int_distribution uid(i64(0), p - 1);
  i64 x, w;
  do {
    x = uid(mt);
    w = (x * x + p - y) % p;
  } while (power(w, (p - 1) / 2, p) == 1);
  auto mul = [&](pair<i64, i64> a, pair<i64, i64> b) {
    return pair((a.first * b.first + a.second * b.second % p * w) % p,
                (a.first * b.second + a.second * b.first) % p);
  };
  pair<i64, i64> a = {x, 1}, res = {1, 0};
  for (i64 r = (p + 1) >> 1; r; r >>= 1, a = mul(a, a)) {
    if (r & 1) { res = mul(res, a); }
  }
  return res.first;
}
```

4.1.2 Logarithm

Find k such that $x^k \equiv y \pmod{n}$.
Constraints: $0 \le x,\, y < n$.

```
i64 log(i64 x, i64 y, i64 n) {
  if (y == 1 or n == 1) { return 0; }
  if (not x) { return y ? -1 : 1; }
  i64 res = 0, k = 1 % n;
  for (i64 d; k != y and (d = gcd(x, n)) != 1; res += 1) {
    if (y % d) { return -1; }
    n /= d;
    y /= d;
    k = k * (x / d) % n;
  }
  if (k == y) { return res; }
  unordered_map<i64, i64> mp;
  i64 px = 1, m = sqrt(n) + 1;
  for (int i = 0; i < m; i += 1, px = px * x % n) { mp[y * px % n] = i; }
  i64 ppx = k * px % n;
  for (int i = 1; i <= m; i += 1, ppx = ppx * px % n) {
    if (mp.count(ppx)) { return res + i * m - mp[ppx]; }
  }
  return -1;
}
```

4.2 Chinese Remainder Theorem

```
tuple<i64, i64, i64> exgcd(i64 a, i64 b) {
  i64 x = 1, y = 0, x1 = 0, y1 = 1;
  while (b) {
    i64 q = a / b;
    tie(x, x1) = pair(x1, x - q * x1);
    tie(y, y1) = pair(y1, x - q * y1);
    tie(a, b) = pair(b, a - q * b);
  }
  return {a, x, y};
}
optional<pair<i64, i64>> linear_equations(i64 a0, i64 b0, i64 a1, i64 b1) {
  auto [d, x, y] = exgcd(a0, a1);
  if ((b1 - b0) % d) { return {}; }
  i64 a = a0 / d * a1, b = (i128)(b1 - b0) / d * x % (a1 / d);
  if (b < 0) { b += a1 / d; }
  b = (i128)(a0 * b + b0) % a;
  if (b < 0) { b += a; }
  return {{a, b}};
}
```

4.3 Miller Rabin

```
bool miller_rabin(i64 n) {
  static constexpr array<int, 9> p = {2, 3, 5, 7, 11, 13, 17, 19, 23};
  if (n == 1) { return false; }
  if (n == 2) { return true; }
  if (not(n % 2)) { return false; }
  int r = countr_zero(u64(n - 1));
  i64 d = (n - 1) >> r;
  for (int pi : p) {
    if (pi >= n) { break; }
    i64 x = power(pi, d, n);
    if (x == 1 or x == n - 1) { continue; };
    for (int j = 1; j < r; j += 1) {
      x = (i128)x * x % n;
      if (x == n - 1) { break; }
    }
    if (x != n - 1) { return false; }
  }
  return true;
};
```

4.4 Pollard Rho

```
vector<i64> pollard_rho(i64 n) {
  static mt19937_64 mt;
  uniform_int_distribution uid(i64(0), n);
  if (n == 1) { return {}; }
  vector<i64> res;
  function<void(i64)> rho = [&](i64 n) {
    if (miller_rabin(n)) { return res.push_back(n); }
    i64 d = n;
    while (d == n) {
      d = 1;
      for (i64 k = 1, y = 0, x = 0, s = 1, c = uid(mt); d == 1;
           k <<= 1, y = x, s = 1) {
        for (int i = 1; i <= k; i += 1) {
          x = ((i128)x * x + c) % n;
          s = (i128)s * abs(x - y) % n;
          if (not(i % 127) or i == k) {
            d = gcd(s, n);
            if (d != 1) { break; }
          }
        }
      }
    }
    rho(d);
    rho(n / d);
  };
  rho(n);
  return res;
}
```

4.5 Primitive Root

Constraints: $n = 2,\, 4,\, p^k,\, 2p^k$ where p is odd prime.

```
i64 phi(i64 n) {
  auto pd = pollard_rho(n);
  ranges::sort(pd);
  pd.erase(ranges::unique(pd).begin(), pd.end());
  for (i64 pi : pd) { n = n / pi * (pi - 1); }
  return n;
}
i64 minimum_primitive_root(i64 n) {
  i64 pn = phi(n);
  auto pd = pollard_rho(pn);
  ranges::sort(pd);
  pd.erase(ranges::unique(pd).begin(), pd.end());
```

```
  auto check = [&](i64 r) {
    if (gcd(r, n) != 1) { return false; }
    for (i64 pi : pd) {
      if (power(r, pn / pi, n) == 1) { return false; }
    }
    return true;
  };
  i64 r = 1;
  while (not check(r)) { r += 1; }
  return r;
}
```

4.6 Sum of Floor

Returns $\sum_{i=0}^{n-1} \lfloor \frac{ai+b}{m} \rfloor$.

```
u64 sum_of_floor(u64 n, u64 m, u64 a, u64 b) {
  u64 ans = 0;
  while (n) {
    if (a >= m) {
      ans += a / m * n * (n - 1) / 2;
      a %= m;
    }
    if (b >= m) {
      ans += b / m * n;
      b %= m;
    }
    u64 y = a * n + b;
    if (y < m) { break; }
    tie(n, m, a, b) = tuple(y / m, a, m, y % m);
  }
  return ans;
}
```

4.7 Minimum of Remainder

Returns $\min\{(ai+b) \bmod m : 0 \le i < n\}$.

```
u64 min_of_mod(u64 n, u64 m, u64 a, u64 b, u64 c = 1, u64 p = 1, u64 q = 1) {
  if (a == 0) { return b; }
  if (c % 2) {
    if (b >= a) {
      u64 t = (m - b + a - 1) / a;
      u64 d = (t - 1) * p + q;
      if (n <= d) { return b; }
      n -= d;
      b += a * t - m;
    }
    b = a - 1 - b;
  } else {
    if (b < m - a) {
      u64 t = (m - b - 1) / a;
      u64 d = t * p;
      if (n <= d) { return (n - 1) / p * a + b; }
      n -= d;
      b += a * t;
    }
    b = m - 1 - b;
  }
  u64 d = m / a;
  u64 res = min_of_mod(n, a, m % a, b, c += 1, (d - 1) * p + q, d * p + q);
  return c % 2 ? m - 1 - res : a - 1 - res;
}
```

4.8 Primes

Minimum prime p s.t. $p = 10^n + k$ for n (A003617).

n	16	17	18
k	10000000000000061	100000000000000003	1000000000000000003

Minimum prime p s.t. $p = k2^n + 1$ for n (A035089).

n	23	26	27	28	31
p	167772161	469762049	2013265921	3221225473	75161927681

5 Numerical

5.1 Golden Search

```
template <int step> f64 golden_search(function<f64(f64)> f, f64 l, f64 r) {
  f64 ml = (numbers::phi - 1) * l + (2 - numbers::phi) * r;
  f64 mr = l + r - ml;
  f64 fml = f(ml), fmr = f(mr);
  for (int i = 0; i < step; i += 1)
    if (fml > fmr) {
      l = ml;
      ml = mr;
      fml = fmr;
      fmr = f(mr = (numbers::phi - 1) * r + (2 - numbers::phi) * l);
    } else {
      r = mr;
      mr = ml;
      fmr = fml;
      fml = f(ml = (numbers::phi - 1) * l + (2 - numbers::phi) * r);
    }
  return midpoint(l, r);
}
```

5.2 Adaptive Simpson

```
f64 simpson(function<f64(f64)> f, f64 l, f64 r) {
  return (r - l) * (f(l) + f(r) + 4 * f(midpoint(l, r))) / 6;
}
f64 adaptive_simpson(const function<f64(f64)> &f, f64 l, f64 r, f64 eps) {
  f64 m = midpoint(l, r);
  f64 s = simpson(f, l, r);
  f64 sl = simpson(f, l, m);
  f64 sr = simpson(f, m, r);
  f64 d = sl + sr - s;
  if (abs(d) < 15 * eps) { return (sl + sr) + d / 15; }
  return adaptive_simpson(f, l, m, eps / 2) +
```

```
        adaptive_simpson(f, m, r, eps / 2);
}
```

5.3 Simplex

Returns maximum of cx s.t. $ax \le b$ and $x \ge 0$.

```
struct Simplex {
  int n, m;
  f64 z;
  vector<vector<f64>> a;
  vector<f64> b, c;
  vector<int> base;
  Simplex(int n, int m)
      : n(n), m(m), a(m, vector<f64>(n)), b(m), c(n), base(n + m), z(0) {
    iota(base.begin(), base.end(), 0);
  }
  void pivot(int out, int in) {
    swap(base[out + n], base[in]);
    f64 f = 1 / a[out][in];
    for (f64 &aij : a[out]) { aij *= f; }
    b[out] *= f;
    a[out][in] = f;
    for (int i = 0; i <= m; i += 1) {
      if (i != out) {
        auto &ai = i == m ? c : a[i];
        f64 &bi = i == m ? z : b[i];
        f64 f = -ai[in];
        if (f < -eps or f > eps) {
          for (int j = 0; j < n; j += 1) { ai[j] += a[out][j] * f; }
          ai[in] = a[out][in] * f;
          bi += b[out] * f;
        }
      }
    }
  }
  bool feasible() {
    while (true) {
      int i = ranges::min_element(b) - b.begin();
      if (b[i] > -eps) { break; }
      int k = -1;
      for (int j = 0; j < n; j += 1) {
        if (a[i][j] < -eps and (k == -1 or base[j] > base[k])) { k = j; }
      }
      if (k == -1) { return false; }
      pivot(i, k);
    }
    return true;
  }
  bool bounded() {
    while (true) {
      int i = ranges::max_element(c) - c.begin();
      if (c[i] < eps) { break; }
      int k = -1;
      for (int j = 0; j < m; j += 1) {
        if (a[j][i] > eps) {
          if (k == -1) {
            k = j;
          } else {
            f64 d = b[j] * a[k][i] - b[k] * a[j][i];
            if (d < -eps or (d < eps and base[j] > base[k])) { k = j; }
          }
        }
      }
      if (k == -1) { return false; }
      pivot(k, i);
    }
    return true;
  }
  vector<f64> x() const {
    vector<f64> res(n);
    for (int i = n; i < n + m; i += 1) {
      if (base[i] < n) { res[base[i]] = b[i - n]; }
    }
    return res;
  }
};
```

5.4 Green's Theorem

$$\oint_C (Pdx + Qdy) = \iint_D \left(\frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}\right) dxdy.$$

5.5 Double Integral

$$\iint_D f(x,y)dxdy = \iint_D f(x(u,v), y(u,v)) \begin{vmatrix} \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \end{vmatrix} dudv.$$

6 Convolution

6.1 $(\mathbb{R}, \times, +)$ on $(\mathbb{Z}, +)$

```
void fft(vector<complex<f64>> &a, bool inverse) {
  int n = a.size();
  vector<int> r(n);
  for (int i = 0; i < n; i += 1) { r[i] = r[i / 2] / 2 | (i % 2 ? n / 2 : 0); }
  for (int i = 0; i < n; i += 1) {
    if (i < r[i]) { swap(a[i], a[r[i]]); }
  }
  for (int m = 1; m < n; m *= 2) {
    complex<f64> wn(cos(numbers::pi / m),
                    sin((inverse ? -1 : 1) * numbers::pi / m));
    for (int i = 0; i < n; i += m * 2) {
      complex<f64> w = 1;
      for (int j = 0; j < m; j += 1, w = w * wn) {
        auto &x = a[i + j + m], &y = a[i + j], t = w * x;
        tie(x, y) = pair(y - t, y + t);
      }
    }
  }
  if (inverse) {
    for (auto &ai : a) { ai /= n; }
  }
}
vector<int> covolution(const vector<int> &a, const vector<int> &b) {
  auto m = a.size() + b.size() - 1;
  auto n = bit_ceil(m);
  vector<complex<f64>> f(n);
  for (int i = 0; i < (int)n; i += 1) {
    f[i] = {i < ssize(a) ? (f64)a[i] : 0., i < ssize(b) ? (f64)b[i] : 0.};
  }
```

```
    fft(f, false);
    for (auto &fi : f) { fi *= fi; }
    fft(f, true);
    vector<int> c(m);
    for (int i = 0; i < (int)m; i += 1) { c[i] = round(f[i].imag() / 2); }
    return c;
}
```

7 Geometry

7.1 Pick's Theorem

$$\text{Area} = \#\{\text{points}\quad\text{inside}\} + \frac{1}{2}\#\{\text{points}\quad\text{on}\quad\text{the}\quad\text{border}\} - 1.$$