Graph & Tree

DSU

Description: Disjoint-set data structure with path compression and union by size. Supports `unite` and `findpar`.
Time: $\mathcal{O}(\alpha(N))$, where α is the inverse Ackermann function ($\approx$ constant).

```
{
public:
  vector<int> parent, size;
  int comp;
  DSU(int n)
  {
    parent.resize(n + 1, 0);
    size.resize(n + 1, 0);
    for (int i = 0; i <= n; i++)
    {
      parent[i] = i;
      size[i] = 1;
    }
    comp = n;
  }
  int findpar(int node)
  {
    if (node == parent[node])
      return node;

    return parent[node] = findpar(parent[
        node]);
  }
  void unite(int u, int v)
  {

    int ulpar_u = findpar(u);
    int ulpar_v = findpar(v);
    if (ulpar_u == ulpar_v)
      return;
    if (size[ulpar_u] < size[ulpar_v])
      swap(ulpar_u, ulpar_v);
    parent[ulpar_v] = ulpar_u;
    size[ulpar_u] += size[ulpar_v];
    comp--;
  }
};
```

SPFA (Shortest Path Faster Algo)

Description: Queue-optimized Bellman-Ford. Computes single-source shortest paths and detects negative cycles.
Time: Average $\mathcal{O}(E)$, Worst $\mathcal{O}(VE)$.

```
vector<int> dis(node + 1, inf);
queue<int> q;
vector<int> count(node + 1, 0);
vector<bool> inqueue(node + 1, false);
dis[1] = 0;
q.push(1);
while (!q.empty())
{
  int node = q.front();
  q.pop();
  inqueue[node] = false;
  for (auto it : graph[node])
  {
    int newnode = it[0];
    int wt = it[1];
    if (dis[newnode] > dis[node] + wt)
    {
      dis[newnode] = dis[node] + wt;
      if (!inqueue[newnode])
      {
        q.push(newnode);
        inqueue[newnode] = true;
        count[newnode]++;
        if (count[newnode] > node)
        {

          cout << "Negative Cycle Found" <<
              endl;
          return;
        }
      }
    }
  }
}
```

Floyd Warshall

Description: All-pairs shortest path algorithm. Works with negative edges (no negative cycles).
Time: $\mathcal{O}(V^3)$.

```
for (int k = 1; k <= nodes; k++) {
    for (int i = 1; i <= nodes; i++) {
        for (int j = 1; j <= nodes; j++) {
            graph[i][j] = min(graph[i][j],
                graph[i][k] + graph[k][j
                ]);
        }
    }
}
```

Dijkstra

Description: Single-source shortest path for non-negative edge weights using a priority queue.
Time: $\mathcal{O}(E \log V)$.

```
priority_queue<array<long long, 2>, vector<
    array<long long, 2>>, greater<array<
    long long, 2>>> pq;
int n = destination + 1;
vector<long long> dist(n, LONG_LONG_MAX);
vector<int> parent(n);
iota(parent.begin(), parent.end(), 0);

pq.push({0, source});
dist[source] = 0;

while (!pq.empty()) {
    int node = pq.top()[1];
    long long wt = pq.top()[0];
    pq.pop();

    if (wt > dist[node]) continue;

    for (auto it : graph[node]) {
        int newnode = it[0];
        long long newwt = it[1];
        if (dist[node] + newwt < dist[
            newnode]) {
            dist[newnode] = dist[node] +
                newwt;
            pq.push({dist[newnode], newnode
                });
            parent[newnode] = node;
        }
    }
}
```

SCC (Kosaraju)

Description: Finds strongly connected components using two DFS passes. Requires `rev[]` (transpose graph).
Time: $\mathcal{O}(V + E)$.

```
// Assume graph[] and rev[] (reverse graph)
    are built
{
  int u, v;
  cin >> u >> v;
  graph[u].pb(v);
  rev[v].pb(u);
}
vector<int> vis(n + 1, 0);
vector<int> order;
auto get = [&](auto &&self, int node) ->
    void
{
  vis[node] = 1;
  for (auto it : graph[node])
  {
    if (vis[it])
      continue;
    self(self, it);
  }

  order.pb(node);
};

for (int i = 1; i <= n; i++)
{
  if (vis[i])
    continue;
  get(get, i);
}
vis.assign(n + 1, 0);
reverse(all(order));
vector<int> cur;
vector<int> comp_id(n + 1, 1);
vector<vector<int>> component;
auto rec = [&](auto &&self, int node, int
    root, int cid) -> void
{
  cur.pb(node);
  comp_id[node] = cid;
  vis[node] = 1;
  for (auto it : rev[node])
  {
    if (vis[it])
      continue;
    self(self, it, root, cid);
  }
};
component.pb({0});
for (auto it : order)
{
  if (vis[it])
    continue;
  int c = component.size();
  rec(rec, it, it, c);
  component.pb(cur);
  cur.clear();
}

int sz = component.size() - 1;
vector<vector<int>> scc(sz + 5);
for (int u = 1; u <= n; u++)
{
  int compu = comp_id[u];
  for (auto v : graph[u])
  {
    int compv = comp_id[v];
    if (compu != compv)
    {
      scc[compu].pb(compv);
    }
  }
}
for (int i = 1; i <= sz; i++)
{
  sort(scc[i].begin(), scc[i].end());
  scc[i].erase(unique(scc[i].begin(), scc[i
      ].end()), scc[i].end());
}
```

LCA (Binary Lifting)

Description: Lowest Common Ancestor using binary lifting. `kth` returns the k-th ancestor.
Time: Build $\mathcal{O}(N \log N)$, Query $\mathcal{O}(\log N)$.

```
int LOG = 1;
while ((1 << LOG) <= n)
  ++LOG;
vector<vector<int>> up(n + 1, vector<int>(
    LOG + 1, 0));
vector<vector<int>> mx(n + 1, vector<int>(
    LOG + 1, 0));
vector<vector<int>> mn(n + 1, vector<int>(
    LOG + 1, 1e9));

auto rec = [&](auto &&self, int node, int
    par, int cur) -> void
{
  parent[node] = par;
  if (par != 0)
    depth[node] = depth[par] + 1;

  up[node][0] = par;
  mx[node][0] = cur;
  mn[node][0] = cur;
  for (int i = 1; i < LOG; i++)
  {
    int prev = up[node][i - 1];
    up[node][i] = up[prev][i - 1];
    mx[node][i] = max(mx[node][i - 1], mx[
        prev][i - 1]);
    mn[node][i] = min(mn[node][i - 1], mn[
        prev][i - 1]);
```

```cpp
  }

  for (auto it : graph[node])
  {
    if (it.ff == par)
      continue;
    self(self, it.ff, node, it.ss);
  }
};

rec(rec, 1, 0, 0);

auto kth = [&](int node, int k) -> array<int, 3>
{
  int mxx = 0, mnn = 1e9;
  for (int i = 0; i < LOG; i++)
  {
    if ((1 << i) & k)
    {
      mxx = max(mxx, mx[node][i]);
      mnn = min(mnn, mn[node][i]);
      node = up[node][i];
    }
  }
  return {node, mnn, mxx};
};
auto lca = [&](int u, int v) -> pair<int, int>
{
  int mxx = 0, mnn = 1e9;
  if (depth[u] > depth[v])
  {
    auto it = kth(u, depth[u] - depth[v]);
    u = it[0];
    mnn = it[1];
    mxx = it[2];
  }
  else if (depth[v] > depth[u])
  {
    auto it = kth(v, depth[v] - depth[u]);
    v = it[0];
    mnn = it[1];
    mxx = it[2];
  }

  if (u == v)
    return {mnn, mxx};

  for (int i = LOG - 1; i >= 0; i--)
  {
    if (up[u][i] != up[v][i])
    {
      mxx = max({mxx, mx[u][i], mx[v][i]});
      mnn = min({mnn, mn[u][i], mn[v][i]});
      u = up[u][i];
      v = up[v][i];
    }
  }
  mxx = max({mxx, mx[u][0], mx[v][0]});
  mnn = min({mnn, mn[u][0], mn[v][0]});
  return {mnn, mxx};
};
```

Centroid Decomposition

Description: Decomposes a tree into a tree of centroids (depth $\mathcal{O}(\log N)$). `update`/`qry` example solves min distance to marked nodes.

Time: Construction $\mathcal{O}(N \log N)$, Queries $\mathcal{O}(\log N)$ or $\mathcal{O}(\log^2 N)$.

```cpp
vector<int> used(n + 1), size(n + 1), parent(n + 1);
vector<int> ans(n + 1, 2e5);
function<int(int, int)> get_size = [&](int node, int par)
{
  size[node] = 1;
  for (auto it : graph[node])
  {
    if (it == par or used[it])
      continue;
    size[node] += get_size(it, node);
  }
  return size[node];
};
function<int(int, int, int)> get_cen = [&](int node, int par, int sz)
{
  for (auto it : graph[node])
  {
    if (it == par or used[it])
      continue;
    if (size[it] > sz / 2)
      return get_cen(it, node, sz);
  }
  return node;
};
function<void(int, int)> decompose = [&](int node, int par)
{
  int sz = get_size(node, 0);
  int cen = get_cen(node, 0, sz);
  used[cen] = 1;
  if (par == 0)
    par = cen;
  parent[cen] = par;
  for (auto it : graph[cen])
  {
    if (used[it])
      continue;
    decompose(it, cen);
  }
};
function<void(int)> update = [&](int cur)
{
  int x = cur;
  ans[cur] = 0;
  while (1)
  {
    ans[x] = min(ans[x], getdis(x, cur));
    if (parent[x] == x)
      break;
    x = parent[x];
  }
};
function<int(int)> qry = [&](int cur)
{
  int x = cur;
  int go = ans[x];
  while (1)
  {
    go = min(go, getdis(x, cur) + ans[x]);
    if (parent[x] == x)
      break;
    x = parent[x];
  }
  return go;
};
decompose(1, 0);
update(1);
while (q--)
{
  int type;
  cin >> type;
  if (type == 1)
  {
    int u;
    cin >> u;
    update(u);
  }
  else
  {
    int u;
    cin >> u;
    cout << qry(u) << el;
  }
}
```

Bridge and Articulation point

Description: Finds Bridges and Articulation Points in an undirected graph using DFS entry times (`tin`) and low-link values (`low`).

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

```cpp
/*
  Finds articulation points and bridges in an undirected simple graph.
  - Nodes are 1..n
  - Input: adjacency list `adj` where adj[u] contains neighbors of u
  - Output:
      vector<int> is_cut(n+1)  : is_cut[u] == 1 if u is an articulation point
      vector<pair<int,int>> bridges : list of bridges (u,v) with u < v
  Usage:
    build adj (size n+1), then call find_articulation_and_bridges(n, adj, is_cut, bridges)
  tin[v] = discovery time of v in DFS.
  low[v] = smallest discovery time reachable from the subtree of v via at most one back edge (i.e., possibly going up to an ancestor).
  If for a child to of v, low[to] > tin[v], then there's no back edge from to's subtree that reaches v or an ancestor of v. So removing the edge (v,to) disconnects the graph $\rightarrow$ a bridge.
  If for a child to of non-root v, low[to] >= tin[v], then removing v disconnects to's subtree from the rest    v is an articulation point. The root is special: it is an articulation point only if it has  2  children in the DFS tree.
*/
void dfs_art_bridge(int v, int p,
const vector<vector<int>> &adj
,vector<int> &tin, vector<int> &low
,vector<int> &is_cut, vector<pair<int, int>> &bridges, int &timer)
{
  tin[v] = low[v] = ++timer;
  int children = 0;
  for (int to : adj[v])
  {
    if (to == p)
      continue; // skip the edge back to parent (simple graph)
    if (tin[to])
    {
      // back edge
      low[v] = min(low[v], tin[to]);
    }
    else
    {
      // tree edge
      ++children;
      dfs_art_bridge(to, v, adj, tin, low, is_cut, bridges, timer);
      low[v] = min(low[v], low[to]);

      // bridge condition (strict)
      if (low[to] > tin[v])
      {
        int a = v, b = to;
        if (a > b)
          swap(a, b);
        bridges.emplace_back(a, b);
      }
      // articulation point (non-root)
      if (p != -1 && low[to] >= tin[v])
      {
        is_cut[v] = 1;
      }
    }
  }
  // root articulation check
  if (p == -1 && children > 1)
    is_cut[v] = 1;
}
void find_articulation_and_bridges
    (int n, const vector<vector<int>> &adj,
          vector<int> &is_cut,
           vector<pair<int, int>> &bridges)
{
  is_cut.assign(n + 1, 0);
  bridges.clear();
  vector<int> tin(n + 1, 0), low(n + 1, 0);
  int timer = 0;
  for (int i = 1; i <= n; ++i) {
    if (!tin[i])
      dfs_art_bridge(i, -1, adj, tin, low,
```

```
        is_cut, bridges, timer);
  }
  sort(bridges.begin(), bridges.end()); // optional: sorted list of bridges
}
int main() {
  vector<int> is_cut;
  vector<pair<int, int>> bridges;
  find_articulation_and_bridges(n, adj, is_cut, bridges);

  // print articulation points
  vector<int> cuts;
  for (int i = 1; i <= n; ++i)
    if (is_cut[i])
      cuts.push_back(i);
  cout << "Articulation points (" << cuts.size() << "):";
  for (int x : cuts)
    cout << ' ' << x;
  cout << '\n';

  // print bridges
  cout << "Bridges (" << bridges.size() << "):\n";
  for (auto &e : bridges)
    cout << e.first << ' ' << e.second << '\n';
}
```

String

Hashing

Description: Double rolling hash using two sets of mods/bases to minimize collisions. Supports $\mathcal{O}(1)$ substring hash queries after $\mathcal{O}(N)$ precomputation. Uses 1-based indexing for queries.

Time: Construction $\mathcal{O}(N)$, Query $\mathcal{O}(1)$.

```
constexpr int mod1 = 1000012253;
constexpr int mod2 = 1000000009;
constexpr int base1=163;
constexpr int base2=271;
template<typename T>
class MultiHashing {
public:
    int n;
    string s;
    string rev;
    vector<pair<T, T>> prefix_hash;
    vector<pair<T, T>> suffix_hash;
    vector<pair<T, T>> power;
    vector<pair<T, T>> inv;
    T mul(T a, T b, T mod) {
        return ((1LL * a % mod) * (b % mod)) % mod;
    }
    T add(T a, T b, T mod) {
        return (1LL * a + b) % mod;
    }
    T sub(T a, T b, T mod) {
        return ((a % mod) - (b % mod) + 2LL * mod) % mod;
    }
    T bigmod(T base, T power, T mod) {
        T res = 1;
        while (power > 0) {
            if (power & 1) {
                res = mul(res, base, mod);
            }
            base = mul(base, base, mod);
            power >>= 1;
        }
        return res;
    }
    MultiHashing(const string& str) : s(str) {
        n = s.size();
        rev = s;
        reverse(rev.begin(), rev.end());
        prefix_hash.resize(n + 1, {0, 0});
        suffix_hash.resize(n + 1, {0, 0});
        power.resize(n + 1, {0, 0});
        inv.resize(n + 1, {0, 0});
        precom();
    }
    void precom() {
        power[0] = {1, 1};
        for (int i = 1; i <= n; i++) {
            power[i].first = mul(power[i - 1].first, base1, mod1);
            power[i].second = mul(power[i - 1].second, base2, mod2);
        }
        T inv_base1 = bigmod(base1, mod1 - 2, mod1);
        T inv_base2 = bigmod(base2, mod2 - 2, mod2);
        inv[0] = {1, 1};
        for (int i = 1; i <= n; i++) {
            inv[i].first = mul(inv[i - 1].first, inv_base1, mod1);
            inv[i].second = mul(inv[i - 1].second, inv_base2, mod2);
        }
        for (int i = 1; i <= n; i++) {
            int ch = s[i - 1] - 'a' + 1;
            prefix_hash[i].first = add(prefix_hash[i - 1].first, mul(ch, power[i - 1].first, mod1), mod1);
            prefix_hash[i].second = add(prefix_hash[i - 1].second, mul(ch, power[i - 1].second, mod2), mod2);
            ch = rev[i - 1] - 'a' + 1;
            suffix_hash[i].first = add(suffix_hash[i - 1].first, mul(ch, power[i - 1].first, mod1), mod1);
            suffix_hash[i].second = add(suffix_hash[i - 1].second, mul(ch, power[i - 1].second, mod2), mod2);

        }
    }
    pair<T, T> get_hash(int l, int r) {
        T val1 = sub(prefix_hash[r].first, prefix_hash[l - 1].first, mod1);
        val1 = mul(val1, inv[l].first, mod1);

        T val2 = sub(prefix_hash[r].second, prefix_hash[l - 1].second, mod2);
        val2 = mul(val2, inv[l].second, mod2);

        return {val1, val2};
    }
    pair<T, T> get_hash_rev(int l, int r) {
        T val1 = sub(suffix_hash[r].first, suffix_hash[l - 1].first, mod1);
        val1 = mul(val1, inv[l].first, mod1);
        T val2= sub(suffix_hash[r].second, suffix_hash[l - 1].second, mod2);
        val2 = mul(val2, inv[l].second, mod2);
        return {val1, val2};
    }
    pair<T, T> combine_hash(pair<T, T> h1, pair<T, T> h2, int l1) {
        T val1 = add(h1.first, mul(h2.first, power[l1].first, mod1), mod1);
        T val2 = add(h1.second, mul(h2.second, power[l1].second, mod2), mod2);
        return {val1, val2};
    }
};
```

Trie

Description: Prefix tree. `insert` adds string, `count` checks existence, `erase` lazily removes. `pref` counts words passing through node, `end` counts words ending at node.

Time: $\mathcal{O}(|S| \cdot \Sigma)$ per operation.

```
class Trie
{
public:
    static const int N=26;
    struct Node
    {
        int next[N];
        int pref;
        int end;
        Node()
        {
            fill(next,next+N,-1);
            pref=0;
            end=0;
        }
    };
    vector<Node> tree;
    Trie(int sz=1)
    {
        tree.reserve(sz);
        tree.emplace_back();
    }
    void insert(string &s)
    {
        int cur=0;
        tree[cur].pref++;
        for(auto it : s)
        {
            int ch=it-'a';
            if(tree[cur].next[ch]==-1)
            {
                tree[cur].next[ch]=(int)tree.size();
                tree.emplace_back();
            }
            cur=tree[cur].next[ch];
            tree[cur].pref++;
        }
        tree[cur].end++;
    }
    int count(string &s)
    {
        int cur=0;
        for(auto it : s)
        {
            int ch=it-'a';
            if(tree[cur].next[ch]==-1)
                return 0;
            cur=tree[cur].next[ch];
        }
        return tree[cur].end;
    }
    int prefixnode(string &s)
    {
        int cur=0;
        for(auto it : s)
        {
            int ch=it-'a';
            if(tree[cur].next[ch]==-1)
                return -1;
            cur=tree[cur].next[ch];
        }
        return cur;
    }
    void erase(string &s)
    {
        if(count(s)==0) return;
        int cur=0;
        tree[cur].pref--;
        for(auto it : s)
        {
            int ch=it-'a';
            cur=tree[cur].next[ch];
            tree[cur].pref--;
        }
        tree[cur].end--;
    }
};
```

Z-Function

Description: `z[i]` is the length of the longest common prefix

between string s and the suffix starting at i.
Time: $\mathcal{O}(N)$.

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

KMP

Description: prefix_function computes $\pi[i]$, the length of the longest proper prefix of $s[0 \dots i]$ that is also a suffix of $s[0 \dots i]$. kmp_search finds all occurrences of pattern.
Time: $\mathcal{O}(N)$.

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

// KMP search: returns starting indices (0-based) of occurrences of pattern in text.
// Time: O(|text| + |pattern|)
vector<int> kmp_search(const string &text, const string &pattern) {
    if (pattern.empty() || text.empty() || pattern.size() > text.size()) return {};
    vector<int> pi = prefix_function(pattern);
    vector<int> matches;
    int j = 0; // number of characters matched in pattern
    for (int i = 0; i < (int)text.size(); ++i) {
        while (j > 0 && text[i] != pattern[j]) j = pi[j-1];
        if (text[i] == pattern[j]) ++j;
        if (j == (int)pattern.size()) {
            matches.push_back(i - j + 1); // match starts at this index (0-based)
            j = pi[j-1]; // continue searching for next match
        }
    }
    return matches;
}
```

Suffix Array

Description: Sorts all suffixes. sa[i] = index of i-th lexicographically smallest suffix. lcp[i] = longest common prefix between sa[i] and sa[i+1].
Time: Build $\mathcal{O}(N \log N)$, LCP $\mathcal{O}(N)$. Pattern search $\mathcal{O}(M \log N)$.

```
// -Radix-style suffix array (doubling, counting/radix sort)
vi build_sa(const string &s) {
    int n = (int)s.size();
    if (n == 0) return {};
    vi sa(n), rankv(n), tmp(n);
    // initial ranks = character codes (0..255)
    for (int i = 0; i < n; ++i) {
        sa[i] = i;
        rankv[i] = (unsigned char)s[i];
    }
    // counting sort by key where keys are in [0..K]
    auto counting_sort_by_key = [&](const vi &sa_in, const vi &key, int K) {
        vi cnt(K + 1);
        for (int i = 0; i < (int)sa_in.size(); ++i) ++cnt[key[sa_in[i]]];
        for (int i = 1; i <= K; ++i) cnt[i] += cnt[i-1];
        vi sa_out(sa_in.size());
        for (int i = (int)sa_in.size() - 1; i >= 0; --i) {
            int x = sa_in[i];
            int k = key[x];
            --cnt[k];
            sa_out[cnt[k]] = x;
        }
        return sa_out;
    };
    for (int k = 1; k < n; k <<= 1) {
        // second key: rank[i+k] (map -1 -> 0, others -> rank+1) so keys are >=0
        vi key2(n);
        int maxKey2 = 0;
        for (int i = 0; i < n; ++i) {
            key2[i] = (i + k < n ? rankv[i + k] + 1 : 0);
            maxKey2 = max(maxKey2, key2[i]);
        }
        sa = counting_sort_by_key(sa, key2, maxKey2);
        // first key: rank[i] (map to rank+1 to keep 0 reserved? not necessary here)
        vi key1(n);
        int maxKey1 = 0;
        for (int i = 0; i < n; ++i) {
            key1[i] = rankv[i] + 1; // ensure keys >=1 (so 0 is reserved for out-of-range second key)
            maxKey1 = max(maxKey1, key1[i]);
        }
        sa = counting_sort_by_key(sa, key1, maxKey1);
        // recompute new ranks
        tmp[sa[0]] = 0;
        for (int i = 1; i < n; ++i) {
            int a = sa[i-1], b = sa[i];
            // compare pairs (rank[a], rank[a+k]) and (rank[b], rank[b+k])
            bool diff = (rankv[a] != rankv[b]) ||
                        ( (a + k < n ? rankv[a + k] : -1) != (b + k < n ? rankv[b + k] : -1) );
            tmp[b] = tmp[a] + (diff ? 1 : 0);
        }
        rankv = tmp;
        if (rankv[sa[n-1]] == n-1) break; // all ranks distinct
    }
    return sa;
}
// Kasai's algorithm to build LCP array from s and sa.
// Returns lcp of size n-1 where lcp[i] = lcp(sa[i], sa[i+1])
vi build_lcp(const string &s, const vi &sa) {
    int n = (int)s.size();
    if (n <= 1) return {};
    vi rank(n), lcp(n - 1);
    for (int i = 0; i < n; ++i) rank[sa[i]] = i;
    int h = 0;
    for (int i = 0; i < n; ++i) {
        int r = rank[i];
        if (r == n - 1) { h = 0; continue; }
        int j = sa[r + 1];
        while (i + h < n && j + h < n && s[i + h] == s[j + h]) ++h;
        lcp[r] = h;
        if (h > 0) --h;
    }
    return lcp;
}
// Count distinct substrings of s using SA+LCP:
long long count_distinct_substrings(const string &s, const vi &sa, const vi &lcp) {
    long long n = s.size();
    long long total = n * (n + 1) / 2;
    long long sum_lcp = 0;
    for (int x : lcp) sum_lcp += x;
    return total - sum_lcp;
}
// Find all occurrences of pattern in text using suffix array.
// Returns pair [L, R) as indices into sa: occurrences are sa[L], sa[L+1], ..., sa[R-1]
pair<int,int> occurrences_range(const string &text, const vi &sa, const string &pattern) {
    int n = (int)text.size();
    int m = (int)pattern.size();
    if (m == 0) return {0, n}; // empty pattern (convention)
    int l = 0, r = n;
    while (l < r) {
        int mid = (l + r) >> 1;
        int cmp = text.compare(sa[mid], m, pattern);
        if (cmp < 0) l = mid + 1;
        else r = mid;
    }
    int L = l;
    l = 0; r = n;
    while (l < r) {
        int mid = (l + r) >> 1;
        int cmp = text.compare(sa[mid], m, pattern);
        if (cmp <= 0) l = mid + 1;
        else r = mid;
    }
    int R = l;
    return {L, R};
}
// Helper: return vector of starting positions (0-based) of all occurrences
vi occurrences(const string &text, const vi &sa, const string &pattern) {
    auto range = occurrences_range(text, sa, pattern);
    vi ans;
    for (int i = range.first; i < range.second; ++i) ans.push_back(sa[i]);
    sort(ans.begin(), ans.end()); // optional: positions in increasing order
    return ans;
}
// Example usage
int main() {
    // Example 1: build SA + LCP and print them
    string s = "banana";
    vi sa = build_sa(s);
    vi lcp = build_lcp(s, sa);
    cout << "String: " << s << "\n";
    cout << "Suffix Array (sa):\n";
    for (int i = 0; i < (int)sa.size(); ++i) {
        cout << i << ": sa=" << sa[i] << " suffix=\"" << s.substr(sa[i]) << "\"\n";
    }
    cout << "LCP array (between sa[i] and sa[i+1]):\n";
```

```
    for (int i = 0; i < (int)lcp.size(); ++i
        ↪ ) {
        cout << "lcp[" << i << "] = " << lcp
            ↪ [i] << "\n";
    }
    // Example 2: count distinct substrings
    cout << "Distinct substrings count = "
        ↪ << count_distinct_substrings(s,
        ↪ sa, lcp) << "\n";
    // Example 3: find occurrences of a
        ↪ pattern
    string pat = "ana";
    vi occ = occurrences(s, sa, pat);
    cout << "Pattern \"" << pat << "\"
        ↪ occurs " << occ.size() << "
        ↪ times at positions (0-based): ";
    for (int p : occ) cout << p << " ";
    cout << "\n";
    // Example 4: longest common substring
        ↪ between two strings (simple use)
    string a = "xabxac";
    string b = "abcabxabcd";
    string T = a + char('#') + b; // '#'
        ↪ must not appear in a or b
    vi sa2 = build_sa(T);
    vi lcp2 = build_lcp(T, sa2);
    int best = 0, pos = -1;
    for (int i = 0; i + 1 < (int)lcp2.size()
        ↪ ; ++i) {
        int x = sa2[i], y = sa2[i+1];
        if ((x < (int)a.size()) != (y < (int
            ↪ )a.size())) {
            if (lcp2[i] > best) { best =
                ↪ lcp2[i]; pos = sa2[i]; }
        }
    }
    cout << "Longest common substring
        ↪ between \"" << a << "\" and \""
        ↪ << b << "\": length=" << best;
    if (best > 0) cout << ", substring=\""
        ↪ << T.substr(pos, best) << "\"";
    cout << "\n";

    return 0;
}
```

Manacher

Description: Computes maximal palindrome lengths. `d1[i]`: max **odd** palindrome centered at i has radius `d1[i]-1`. `d2[i]`: max **even** palindrome centered at i has radius `d2[i]-1`.

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

```
n = sz(s);
vector<ll> d1(n); // maximum odd length
    ↪ palindrome centered at i
// here d1[i]=the palindrome has d1[i]-1
    ↪ right characters from i
// e.g. for aba, d1[1]=2;
for (i = 0, l = 0, r = -1; i < n; i++)
{
  k = (i > r) ? 1 : min(d1[l + r - i], r - i
      ↪ );
  while (0 <= i - k && i + k < n && s[i - k]
      ↪ == s[i + k])
  {
    k++;
  }
  d1[i] = k--;
  if (i + k > r)
  {
    l = i - k;
    r = i + k;
  }
}
vector<ll> d2(n); // maximum even length
    ↪ palindrome centered at i
// here d2[i]=the palindrome has d2[i]-1
    ↪ right characters from i
// e.g. for abba, d2[2]=2;
for (i = 0, l = 0, r = -1; i < n; i++)
{
  k = (i > r) ? 0 : min(d2[l + r - i + 1], r
      ↪ - i + 1);
  while (0 <= i - k - 1 && i + k < n && s[i
      ↪ - k - 1] == s[i + k])
  {
    k++;
  }
  d2[i] = k--;
  if (i + k > r)
  {
    l = i - k - 1;
    r = i + k;
  }
}
```

Data Structure

Sparse Table

Description: Static Range Minimum Query (RMQ). `query` is idempotent ($\mathcal{O}(1)$), `query1` is cascading for non-idempotent functions ($\mathcal{O}(\log N)$).

Time: Build $\mathcal{O}(N \log N)$, Query $\mathcal{O}(1)$.

```
template <typename T>
class SparseTable
{
    public:
    vector<vector<T> > st;
    T op(T a,T b)
    {
        return max(a,b);
    }
    SparseTable(int n,vector<T> &vec)
    {
        st.resize(n+2,vector<T> (__lg(n)+2))
            ↪ ;
        for(int i=1;i<=n;i++)
        {
            st[i][0]=vec[i];
        }
        int k=__lg(n)+1;
        for(int j=1;j<=k;j++)
        {
            for(int i=1;i+(1<<j)<=n+1;i++)
            {
                st[i][j]=op(st[i][j-1],st[i
                    ↪ +(1<<(j-1))][j-1]);
            }
        }
    }
    T query(int l,int r)
    {
        int j=__lg(r-l+1);
        return op(st[l][j],st[r-(1<<j)+1][j
            ↪ ]);
    }
    T query1(int l,int r)
    {
        int ans=0;
        for(int j=__lg(r-l+1);j>=0;j--)
        {
            if((1<<j)<=(r-l+1))
            {
                ans=op(ans,st[l][j]);
                l+=(1<<j);
            }
        }
        return ans;
    }
};
```

BIT 1D

Description: Point update, Prefix sum. `lower_bound` finds smallest index i such that sum$(1 \ldots i) \geq$ val (requires non-negative values).

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

```
struct BIT {
  int n;
  vector<long long> bit;
  BIT(int n=0){ init(n); }
  void init(int _n){
      n = _n;
      bit.assign(n+1, 0);
  }
  // add value `delta` at index i (1-based)
  void add(int i, long long delta){
      for (; i <= n; i += i & -i) bit[i] +=
          ↪ delta;
  }
  // prefix sum [1..i] (1-based)
  long long sumPrefix(int i){
      long long s = 0;
      for (; i > 0; i -= i & -i) s += bit[i
          ↪ ];
      return s;
  }
  // range sum [l..r], 1-based
  long long sumRange(int l, int r){
      if (r < l) return 0;
      return sumPrefix(r) - sumPrefix(l-1);
  }
  // find smallest index idx such that
      ↪ sumPrefix(idx) >= value (value >=
      ↪ 1)
  // returns n+1 if not found
  int lower_bound(long long value){
      if (value <= 0) return 1;
      int idx = 0;
      int bitMask = 1;
      while (bitMask << 1 <= n) bitMask <<=
          ↪ 1;
      for (int k = bitMask; k; k >>= 1){
          int next = idx + k;
          if (next <= n && bit[next] < value
              ↪ ){
              idx = next;
              value -= bit[next];
          }
      }
      return idx + 1;
  }
};
```

BIT 2D

Description: 2D Fenwick Tree for point updates and rectangle sums. 1-based indexing.

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

```
struct BIT2D {
  int n, m;
  vector<vector<long long>> bit;
  BIT2D(int _n=0, int _m=0){ init(_n,_m); }
  void init(int _n, int _m){
      n = _n; m = _m;
      bit.assign(n+1, vector<long long>(m+1,
          ↪ 0));
  }
  // point add at (x,y) (1-based)
  void add(int x, int y, long long delta){
      for (int i = x; i <= n; i += i & -i)
          for (int j = y; j <= m; j += j & -
              ↪ j)
              bit[i][j] += delta;
  }
  // prefix sum (1..x, 1..y)
  long long sumPrefix(int x, int y){
      long long res = 0;
      for (int i = x; i > 0; i -= i & -i)
          for (int j = y; j > 0; j -= j & -j
              ↪ )
              res += bit[i][j];
      return res;
  }
  // rectangle sum (x1,y1) .. (x2,y2),
      ↪ inclusive, 1-based
  long long range_sum(int x1, int y1, int x2
      ↪ , int y2){
      if (x2 < x1 || y2 < y1) return 0;
      return sumPrefix(x2, y2) - sumPrefix(
          ↪ x1-1, y2)
             - sumPrefix(x2, y1-1) +
                 ↪ sumPrefix(x1-1, y1-1);
  }
};
```

MO's Algorithm (Hilbert)

Description: Offline range query processing using Hilbert Curve order to improve cache locality and reduce movement. Significantly faster than standard block sorting.

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

```
class dat
{
public:
```

```cpp
  int l, r, id;
  dat() {};
  dat(int l, int r, int id)
  {
    this->l = l;
    this->r = r;
    this->id = id;
  }
};
void solve()
{
  int n;
  cin >> n;
  vector<int> vec(n);
  for (int i = 0; i < n; i++)
  {
    cin >> vec[i];
  }
  int dis = 0;
  vector<int> freq(1e6 + 5, 0);
  int block_size = sqrt(n);
  int q;
  cin >> q;
  vector<dat> query(q);
  for (int i = 0; i < q; i++)
  {
    int l, r;
    cin >> l >> r;
    l--, r--;
    query[i] = dat(l, r, i);
  }
  auto hilbertorder = [&](int x, int y) -> long long
  {
    const int LOG = 21;
    long long d = 0;
    for (int s = 1 << (LOG - 1); s; s >>= 1)
    {
      bool rx = x & s, ry = y & s;
      d = (d << 2) | (rx * 3 ^ static_cast<int>(ry));
      if (!ry)
      {
        if (rx)
        {
          x = (1 << LOG) - 1 - x;
          y = (1 << LOG) - 1 - y;
        }
        swap(x, y);
      }
    }
    return d;
  };
  vector<pair<long long, int>> order(q);
  for (int i = 0; i < q; i++)
  {
    order[i] = {hilbertorder(query[i].l, query[i].r), i};
  }
  sort(order.begin(), order.end());
  vector<dat> sorted;
  sorted.reserve(q);
  for (auto [_, idx] : order)
    sorted.push_back(query[idx]);
  query.swap(sorted);
  vector<int> ans(q);
  auto add = [&](int ind)
  {
    freq[vec[ind]]++;
    if (freq[vec[ind]] == 1)
      dis++;
  };
  auto remove = [&](int ind)
  {
    freq[vec[ind]]--;
    if (freq[vec[ind]] == 0)
      dis--;
  };
  int L = 0, R = -1;
  for (int i = 0; i < q; i++)
  {
    int l = query[i].l;
    int r = query[i].r;
    int id = query[i].id;
    while (L > l)
    {
      add(--L);
    }
    while (R < r)
    {
      add(++R);
    }
    while (L < l)
    {
      remove(L++);
    }
    while (R > r)
    {
      remove(R--);
    }
    ans[id] = dis;
  }

  for (int i = 0; i < q; i++)
  {
    cout << ans[i] << el;
  }
}
```

Merge Sort Tree

```cpp
class node
{
public:
    vector<int> v;
    vector<ll> pref;
    node(){};
    node(int x)
    {
        v.pb(x);
        pref.resize(1,0);
        pref[0]=x;
    }
};
template <typename Node=node>
class SegmentTree
{
public:
    vector<Node> st;
    Node op(Node &a,Node &b)
    {
        node cur;
        int sz=a.v.size()+b.v.size();
        cur.v.resize(sz,0);
        cur.pref.resize(sz,0);
        merge(all(a.v),all(b.v),cur.v.begin());
        cur.pref[0]=cur.v[0];
        for(int i=1;i<sz;i++)
        {
            cur.pref[i]=cur.v[i]+cur.pref[i-1];
        }
        return cur;
    }
    SegmentTree(vector<int> &vec, int n)
    {
        st.resize(4*n,Node());
        function<void(int, int, int)> build = [&](int id, int start, int end)
        {
            if (start == end)
            {
                st[id]=Node(vec[start]);
                return;
            }
            int mid = (start + end) / 2;
            build(2 * id, start, mid);
            build(2 * id + 1, mid + 1, end);
            st[id] = op(st[2*id],st[2*id+1]);
        };
        build(1, 1, n);
    }
    ll query(int id, int start, int end,ll l,ll r,ll k)
    {
        if (start > r or end < l)
            return 0;
        if (start >= l and end <= r)
        {
            auto lo=upper_bound(all(st[id].v),k);
            int ind=lo-st[id].v.begin();
            if(ind==0) return 0;
            return st[id].pref[ind-1];
        };
        ll mid = start + (end - start) / 2;
        ll left = query(2 * id, start, mid, l, r,k);
        ll right = query(2 * id + 1, mid + 1, end, l, r,k);
        return (left+right);
    }
};
```

XOR Trie

Description: Binary Trie for integers. Supports finding pair with maximum XOR.

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

```cpp
class TrieNode
{
public:
  TrieNode *left;
  TrieNode *right;
  int cnt = 0;
  TrieNode()
  {
    left = NULL;
    right = NULL;
    cnt = 0;
  }
};
class Trie
{
  TrieNode *root;

public:
  Trie()
  {
    root = new TrieNode();
  }
  void insert(int n)
  {
    TrieNode *curr = root;
    for (int i = 31; i >= 0; i--)
    {
      int bit = (1 & (n >> i));
      if (bit == 0)
      {
        if (curr->left == NULL)
        {
          curr->left = new TrieNode();
        }
        curr = curr->left;
        curr->cnt++;
      }
      else
      {
        if (curr->right == NULL)
        {
          curr->right = new TrieNode();
        }
        curr = curr->right;
        curr->cnt++;
      }
    }
  }
  void remove(int n)
  {
    TrieNode *curr = root;
    for (int i = 31; i >= 0; i--)
    {
      if (curr == NULL)
        break;
      int bit = (n >> i) & 1;
      if (bit == 0)
      {
        curr = curr->left;
        curr->cnt--;
      }
      else
      {
        curr = curr->right;
        curr->cnt--;
      }
    }
```

```cpp
  }
  int max_xor_pair(int n)
  {
    TrieNode *curr = root;
    int ans = 0;
    for (int i = 31; i >= 0; i--)
    {
      if (curr == NULL)
      {
        break;
      }
      int bit = (1 & (n >> i));
      if (bit == 0)
      {
        if (curr->right != NULL and curr->right->cnt > 0)
        {
          ans += (1 << i);
          curr = curr->right;
        }
        else
          curr = curr->left;
      }
      else
      {
        if (curr->left != NULL and curr->left->cnt > 0)
        {
          ans += (1 << i);
          curr = curr->left;
        }
        else
          curr = curr->right;
      }
    }
    return ans;
  }
};
```

LAZY SegTree

Description: Standard Lazy Propagation for range updates.
Time: $\mathcal{O}(\log N)$.

```cpp
struct ST{
    int n;
    vector<int> t,lazy,arr;
    void init(int n) {
        this->n=n;
        t.assign(3*n+5,0);
        lazy.assign(3*n+5,0);
        arr.assign(n+5,0);
    }
    inline void push(int node,int l,int r){
        if(!lazy[node]) return;
        t[node]+=lazy[node]*(r-l+1); // check here
        if(l!=r){
            lazy[node*2]+=lazy[node];
            lazy[node*2+1]+=lazy[node];
        }
        lazy[node]=0;
    }
    inline void here(int node){
        t[node]=t[node*2]+t[node*2+1];  // check here
    }
    void build(int node,int l,int r){
        lazy[node]=0;
        if(l==r){
            t[node]=arr[l];
            return;
        }
        ll mid=(l+r)>>1;
        build(node*2,l,mid);
        build(node*2+1,mid+1,r);
        here(node);
    }
    void upd(int node,int l,int r,int i,int j,int value){
        push(node,l,r);
        if(l>j || r<i) return;
        if(i<=l && r<=j){
            lazy[node]+=value; // check here
            push(node,l,r);
            return;
        }
        ll mid=(l+r)>>1;
        upd(node*2,l,mid,i,j,value);
        upd(node*2+1,mid+1,r,i,j,value);
        here(node);
    }
    ll query(int node,int l,int r,int i,int j){
        push(node,l,r);
        if(l>j || r<i) return 0; /// check here
        if(i<=l && r<=j) return t[node];
        ll mid=(l+r)>>1;
        return query(node*2,l,mid,i,j)+query(node*2+1,mid+1,r,i,j); // check here
    }
}t;
```

PST (Persistent SegTree)

Description: Persistent segment tree. `add_copy` branches off a version.
Time: $\mathcal{O}(\log N)$ query/update. **Space:** $\mathcal{O}(Q \log N)$.

```cpp
class PST{
    private:
        struct node{
            ll sum=0;
            int lc=0,rc=0; // left child right child
        };
    const int n;
    vector<node> tree;
    int timer=1;
    node join(int lc,int rc){
        return node{tree[lc].sum+tree[rc].sum,lc,rc}; // check here
    }
    int build_(int l,int r,const vector<int> &arr){
        int id=timer++;
        if(l==r){
            tree[id]={arr[l],0,0}; // check here
            return id;
        }
        int mid=(l+r)>>1;
        tree[id]=join(build_(l,mid,arr),build_(mid+1,r,arr));
        return id;
    }
    int upd_(int v,int l,int r,int pos,int val){
        int id=timer++;
        if(l==r){
            tree[id]={val,0,0}; // check here
            return id;
        }
        int mid=(l+r)>>1;
        if(pos<=mid) tree[id]=join(upd_(tree[v].lc,l,mid,pos,val),tree[v].rc);
        else tree[id]=join(tree[v].lc,upd_(tree[v].rc,mid+1,r,pos,val));
        return id;
    }
    ll query_(int v,int l,int r,int i,int j){
        if(l>j || r<i) return 0LL; /// check here
        if(i<=l && r<=j) return tree[v].sum;
        int mid=(l+r)>>1;
        return query_(tree[v].lc,l,mid,i,j)+query_(tree[v].rc,mid+1,r,i,j);
    }
    public:
    PST(int n,int mx_nodes) : n(n),tree(mx_nodes) {}
    int build(const vector<int> &arr) { return build_(1,n,arr); }
    int upd(int root,int pos,int val) { return upd_(root,1,n,pos,val); }
    ll query(int root,int l,int r) { return query_(root,1,n,l,r); }
    int add_copy(int root){
        tree[timer]=tree[root];
        return timer++;
    }
};
int32_t main()
{
    const int mx_nodes=2*n+q*(2+__lg(n));
    PST t(n,mx_nodes);
    vector<int> roots = {t.build(a)};
    while(q--){
        int type,k; cin>>type>>k;
        k--;
        if(type==1){
            int pos,val; cin>>pos>>val;
            roots[k]=t.upd(roots[k],pos,val);
        }else if(type==2){
            int a,b; cin>>a>>b;
            cout<<t.query(roots[k],a,b)<<endl;
        }else{
            roots.PB(t.add_copy(roots[k]));
        }
    }
}
```

Dynamic SegTree

Description: Segment tree with sparse coordinates ($N \approx 10^9$). Nodes created on demand.
Time: $\mathcal{O}(\log(\text{Range}))$.

```cpp
class SparseSegTree {
private:
    struct node {
        ll freq=0;
        ll lazy=0;
        int left=0;
        int right=0;
        bool lazy_flag=false;
    };
    vector<node> tree;
    const ll n;
    int timer=1;
    // int comb(int a,int b) { return a+b; }
    void apply(int cur,ll l,ll r,ll val) { // check here
        tree[cur].lazy=val;
        tree[cur].lazy_flag=true;
        tree[cur].freq=(r-l+1)*val;
    }
    void push_down(int cur,ll l,ll r){
        if(!tree[cur].left){
            tree[cur].left= ++timer;
            tree.PB(node());
        }
        if(!tree[cur].right){
            tree[cur].right= ++timer;
            tree.PB(node());
        }
        if(!tree[cur].lazy_flag) return;
        ll mid=(l+r)>>1;
        apply(tree[cur].left,l,mid,tree[cur].lazy);
        apply(tree[cur].right,mid+1,r,tree[cur].lazy);
        tree[cur].lazy_flag=false;
        tree[cur].lazy=0;
    }
    void upd(int cur,ll l,ll r,ll ql,ll qr,ll val) {
        if(qr<l || ql>r) return;
        if(ql<=l && r<=qr) apply(cur,l,r,val);
        else {
            push_down(cur,l,r);
            ll mid=(l+r)>>1;
            upd(tree[cur].left,l,mid,ql,qr,val);
            upd(tree[cur].right,mid+1,r,ql,qr,val);
            tree[cur].freq=
                tree[tree[cur].left].freq + tree[tree[cur].right].freq; // check here
```

```
        }
    }
    ll query(int cur,ll l,ll r,ll ql,ll qr) {
        if(qr<l || ql>r || !cur) return 0;
        if(ql<=l && r<=qr) return tree[cur].freq;
        push_down(cur,l,r);
        ll mid=(l+r)>>1;
        return query(tree[cur].left,l,mid,ql,qr) +
               query(tree[cur].right,mid+1,r,ql,qr); // check here
    }
public:
    SparseSegTree(ll n,int q=0) : n(n) {
        if(q>0) { tree.reserve(2*q*__lg(n)); }
        tree.PB(node()); tree.PB(node());
    }
    void upd(ll ql,ll qr,ll val) { upd(1,1,n,ql,qr,val); }
    int query(ll ql,ll qr) {return query(1,1,n,ql,qr); }
};
int32_t main(){
    const int range_size=1e9;
    SparseSegTree st(range_size+1,q); // pass n+q if there is n given
}
```

Wavelet Tree

Description: Partitions array based on values. `kth`: k-th smallest in range. `LTE`: count values $\leq k$. `count`: range value freq.

Time: $\mathcal{O}(\log(\max A))$ per query.

```
struct wavelet_tree
{
  int lo, hi;
  wavelet_tree *l, *r;
  vi b;
  wavelet_tree(int *from, int *to, int x, int y)
  {
    lo = x, hi = y;
    if (lo == hi or from >= to)
      return;
    int mid = (lo + hi) / 2;
    auto f = [mid](int x)
    {
      return x <= mid;
    };
    b.reserve(to - from + 1);
    b.pb(0);
    for (auto it = from; it != to; it++)
      b.pb(b.back() + f(*it));
    // see how lambda function is used here
    auto pivot = stable_partition(from, to, f);
    l = new wavelet_tree(from, pivot, lo, mid);
    r = new wavelet_tree(pivot, to, mid + 1, hi);
  }
  // kth smallest element in [l, r]
  int kth(int l, int r, int k)
  {
    if (l > r)
      return 0;
    if (lo == hi)
      return lo;
    int inLeft = b[r] - b[l - 1];
    int lb = b[l - 1]; // amt of nos in first (l-1) nos that go in left
    int rb = b[r];     // amt of nos in first (r) nos that go in left
    if (k <= inLeft)
      return this->l->kth(lb + 1, rb, k);
    return this->r->kth(l - lb, r - rb, k - inLeft);
  }
  // count of nos in [l, r] Less than or equal to k
  int LTE(int l, int r, int k)
  {
    if (l > r or k < lo)
      return 0;
    if (hi <= k)
      return r - l + 1;
    int lb = b[l - 1], rb = b[r];
    return this->l->LTE(lb + 1, rb, k) + this->r->LTE(l - lb, r - rb, k);
  }
  int count(int l, int r, int k)
  {
    if (l > r or k < lo or k > hi)
      return 0;
    if (lo == hi)
      return r - l + 1;
    int lb = b[l - 1], rb = b[r], mid = (lo + hi) / 2;
    if (k <= mid)
      return this->l->count(lb + 1, rb, k);
    return this->r->count(l - lb, r - rb, k);
  }
  ~wavelet_tree()
  {
    delete l;
    delete r;
  }
};
int main()
{
  wavelet_tree T(a + 1, a + n + 1, 1, MAX);
}
```

SEGTree Beats (main)

Description: "Jiry Match" Tree. Supports Range Chmin ($a_i = \min(a_i, x)$), Chmax, Add, Set, Mod, Divide, Negative. Handles history/break conditions.

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

```
#include"bits/stdc++.h"
using namespace std;
using ll=long long;
#define endl '\n'
const ll INF=1e18;
const ll NINF=-1e18;
struct STBeats {
private:
    struct node{
        ll max1;              // max value
        ll max2;              // second max value
        int max_cnt;          // cnt of the largest value
        ll min1;              // min value
        ll min2;              // second min value
        int min_cnt;          // cnt of the smallest value
        ll sum;               // sum of the range
        int len;              // length of the range
        ll lazy_add;          // lazy teg
        ll lazy_set;
        bool lazy_neg;
        node() : max1(NINF),max2(NINF),max_cnt(0),
                 min1(INF),min2(INF),min_cnt(0),sum(0),len(0),
                 lazy_add(0),lazy_set(INF),lazy_neg(false) {}
    };

    int n;
    vector<node> tree;
    inline node merge(const node& left, const node& right) {        // O(1)
        node res;
        res.sum=left.sum+right.sum;
        res.len=left.len+right.len;
        res.lazy_add=0;
        res.lazy_set=INF;
        res.lazy_neg=false;
        if(left.max1>right.max1) { // merging max data for chmin
            res.max1=left.max1;
            res.max2=max(left.max2,right.max1);
            res.max_cnt=left.max_cnt;
        }else if(left.max1<right.max1) {
            res.max1=right.max1;
            res.max2=max(left.max1,right.max2);
            res.max_cnt=right.max_cnt;
        }else if(left.max1==right.max1) {
            res.max1=left.max1;
            res.max2=max(left.max2,right.max2);
            res.max_cnt=left.max_cnt+right.max_cnt;
        }

        if(left.min1<right.min1) {  // margin min data for chmax
            res.min1=left.min1;
            res.min2=min(left.min2,right.min1);
            res.min_cnt=left.min_cnt;
        }else if(left.min1>right.min1) {
            res.min1=right.min1;
            res.min2=min(left.min1,right.min2);
            res.min_cnt=right.min_cnt;
        }else if(left.min1==right.min1) {
            res.min1=left.min1;
            res.min2=min(left.min2,right.min2);
            res.min_cnt=left.min_cnt+right.min_cnt;
        }
        return res;
    }
    inline void apply_negative(int v) {
        swap(tree[v].max1,tree[v].min1);
        swap(tree[v].max2,tree[v].min2);
        swap(tree[v].max_cnt,tree[v].min_cnt);
        tree[v].max1*=-1;
        if(tree[v].max2!=NINF) tree[v].max2*=-1;
        tree[v].min1*=-1;
        if(tree[v].min2!=INF) tree[v].min2*=-1;
        tree[v].sum*=-1;
        if(tree[v].lazy_set!=INF) tree[v].lazy_set*=-1;
        else tree[v].lazy_add*=-1;
        tree[v].lazy_neg^=1;
    }
    inline void apply_add(int v,ll x) {                // O(1)
        if(!x) return;
        tree[v].sum+=tree[v].len*x;
        tree[v].max1+=x;
        if(tree[v].max2!=NINF) tree[v].max2+=x;
        tree[v].min1+=x;
        if(tree[v].min2!=INF) tree[v].min2+=x;

        if(tree[v].lazy_set!=INF) tree[v].lazy_set+=x;
        else tree[v].lazy_add+=x;
    }

    inline void apply_set(int v,ll x) {
        tree[v].max1=x;
        tree[v].max2=NINF;
        tree[v].max_cnt=tree[v].len;
        tree[v].min1=x;
        tree[v].min2=INF;
        tree[v].min_cnt=tree[v].len;
        tree[v].sum=tree[v].len*x;
        tree[v].lazy_add=0;
        tree[v].lazy_set=x;
        tree[v].lazy_neg=false;
    }
    inline void apply_chmin(int v,ll x) {              // O(1)
        if(x>=tree[v].max1) return;
        tree[v].sum-=tree[v].max_cnt*(tree[v
```

```cpp
        ].max1-x);
        if(tree[v].min1==tree[v].max1) tree[v].min1=x;
        if(tree[v].min2==tree[v].max1) tree[v].min2=x;
        tree[v].max1=x;

        if(tree[v].lazy_set !=INF)
            tree[v].lazy_set=min(tree[v].lazy_set,x);
    }

    inline void apply_chmax(int v,ll x) {            // O(1)
        if(x<=tree[v].min1) return;
        tree[v].sum+=tree[v].min_cnt*(x-tree[v].min1);
        if(tree[v].max1==tree[v].min1) tree[v].max1=x;
        if(tree[v].max2==tree[v].min1) tree[v].max2=x;
        tree[v].min1=x;

        if(tree[v].lazy_set !=INF)
            tree[v].lazy_set=max(tree[v].lazy_set,x);
    }

    void push_lazy(int v,int tl,int tr) {             // O(1)
        if(tl==tr) return;
        if(tree[v].lazy_set!=INF) {
            apply_set(2*v,tree[v].lazy_set);
            apply_set(2*v+1,tree[v].lazy_set);
            tree[v].lazy_set=INF;
            return;
        }
        if(tree[v].lazy_neg) {
            apply_negative(2*v);
            apply_negative(2*v+1);
            tree[v].lazy_neg=false;
        }
        if(tree[v].lazy_add!=0) {        // for lazy add
            apply_add(2*v,tree[v].lazy_add);
            apply_add(2*v+1,tree[v].lazy_add);
            tree[v].lazy_add=0;
        }
    }
    void push_beats(int v,int tl,int tr) {
        if(tl==tr) return;
        apply_chmin(2*v,tree[v].max1);
        apply_chmin(2*v+1,tree[v].max1);
        apply_chmax(2*v,tree[v].min1);
        apply_chmax(2*v+1,tree[v].min1);
    }

    void build_(int v,int tl,int tr,const vector<ll>& a) {        // O(n)
        if(tl==tr){
            tree[v].len=1;
            tree[v].sum=a[tl];
            tree[v].max1=a[tl];
            tree[v].max_cnt=1;
            tree[v].max2=NINF;
            tree[v].min1=a[tl];
            tree[v].min_cnt=1;
            tree[v].min2=INF;
            tree[v].lazy_add=0;
            tree[v].lazy_set=INF;
            tree[v].lazy_neg=false;
        }else{
            int mid=(tl+tr)>>1;
            build_(2*v,tl,mid,a);
            build_(2*v+1,mid+1,tr,a);
            tree[v]=merge(tree[2*v],tree[2*v+1]);
        }
    }

    void upd_min_(int v,int tl,int tr,int ql,int qr,ll x) {   // O(log^2 n)
        push_lazy(v,tl,tr);
        if(tree[v].max1<=x || qr<tl || tr<ql) return;
        if(ql<=tl && tr<=qr && tree[v].max2<x) {
            apply_chmin(v,x);
            return;
        }
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        upd_min_(2*v,tl,mid,ql,qr,x);
        upd_min_(2*v+1,mid+1,tr,ql,qr,x);
        tree[v]=merge(tree[2*v],tree[2*v+1]);
    }

    void upd_max_(int v,int tl,int tr,int ql,int qr,ll x) {   // O(log^2 n)
        push_lazy(v,tl,tr);
        if(tree[v].min1>=x || qr<tl || tr<ql) return;
        if(ql<=tl && tr<=qr && tree[v].min2>x) {
            apply_chmax(v,x);
            return;
        }
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        upd_max_(2*v,tl,mid,ql,qr,x);
        upd_max_(2*v+1,mid+1,tr,ql,qr,x);
        tree[v]=merge(tree[2*v],tree[2*v+1]);
    }

    void upd_add_(int v,int tl,int tr,int ql,int qr,ll x) {   // O(log n)
        if(qr<tl || tr<ql) return;
        if(ql<=tl && tr<=qr) {
            apply_add(v,x);
            return;
        }
        push_lazy(v,tl,tr);
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        upd_add_(2*v,tl,mid,ql,qr,x);
        upd_add_(2*v+1,mid+1,tr,ql,qr,x);
        tree[v]=merge(tree[2*v],tree[2*v+1]);
    }

    void upd_set_(int v,int tl,int tr,int ql,int qr,ll x) {   // O(log n) range set
        if(qr<tl || tr<ql) return;
        if(ql<=tl && tr<=qr) {
            apply_set(v,x);
            return;
        }
        push_lazy(v,tl,tr);
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        upd_set_(2*v,tl,mid,ql,qr,x);
        upd_set_(2*v+1,mid+1,tr,ql,qr,x);
        tree[v]=merge(tree[2*v],tree[2*v+1]);
    }

    void upd_mod_(int v,int tl,int tr,int ql,int qr,ll x) {   // O(log^2 n)
        push_lazy(v,tl,tr);
        if(tree[v].max1<x || qr<tl || tr<ql) return;
        if(tl==tr) {
            apply_set(v,tree[v].sum%x);
            return;
        }
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        upd_mod_(2*v,tl,mid,ql,qr,x);
        upd_mod_(2*v+1,mid+1,tr,ql,qr,x);
        tree[v]=merge(tree[2*v],tree[2*v+1]);
    }
    ll floor_div(ll a,ll b) {
        if(b<0) { a=-a,b=-b; }
        ll d=a/b;
        ll r=a%b;
        if(r<0) return d-1;
        return d;
    }
    void upd_negative_(int v,int tl,int tr,int ql,int qr) {
        if(qr<tl || tr<ql) return;
        if(ql<=tl && tr<=qr) {
            apply_negative(v);
            return;
        }
        push_lazy(v,tl,tr);
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        upd_negative_(2*v,tl,mid,ql,qr);
        upd_negative_(2*v+1,mid+1,tr,ql,qr);
        tree[v]=merge(tree[2*v],tree[2*v+1]);
    }
    void upd_divide_(int v,int tl,int tr,int ql,int qr,ll x) {    // O(log^2 n)
        if(x==1) return;
        if(x==-1){
            upd_negative_(v,tl,tr,ql,qr);
            return;
        }
        if(qr<tl || tr<ql || !x) return;
        push_lazy(v,tl,tr);
        ll new_min=floor_div(tree[v].min1,x);
        ll new_max=floor_div(tree[v].max1,x);
        if(ql<=tl && tr<=qr && new_min==new_max){
            apply_set(v,new_min);
            return;
        }
        if(tl==tr) {
            ll val=floor_div(tree[v].sum,x);
            apply_set(v,val);
            return;
        }
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        upd_divide_(2*v,tl,mid,ql,qr,x);
        upd_divide_(2*v+1,mid+1,tr,ql,qr,x);
        tree[v]=merge(tree[2*v],tree[2*v+1]);
    }

    ll query_sum_(int v,int tl,int tr,int ql,int qr) { // O(log n)
        if(qr<tl || tr<ql) return 0;
        if(ql<=tl && tr<=qr) return tree[v].sum;
        push_lazy(v,tl,tr);
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        return query_sum_(2*v,tl,mid,ql,qr) + query_sum_(2*v+1,mid+1,tr,ql,qr);
    }
    ll query_max_(int v,int tl,int tr,int ql,int qr) { // O(log n)
        if(qr<tl || tr<ql) return NINF;
        if(ql<=tl && tr<=qr) return tree[v].max1;
        push_lazy(v,tl,tr);
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        return max(query_max_(2*v,tl,mid,ql,qr) , query_max_(2*v+1,mid+1,tr,ql,qr));
    }
    ll query_min_(int v,int tl,int tr,int ql,int qr) { // O(log n)
        if(qr<tl || tr<ql) return INF;
        if(ql<=tl && tr<=qr) return tree[v].min1;
        push_lazy(v,tl,tr);
        push_beats(v,tl,tr);
        int mid=(tl+tr)>>1;
        return min(query_min_(2*v,tl,mid,ql,qr) , query_min_(2*v+1,mid+1,tr,ql,qr));
    }

public:
    STBeats(int n_val) : n(n_val) { tree.
```

```
        resize(4*n+4); }
    void build(const vector<ll>& a) { build_(1,1,n,a); }
    void upd_min(int ql,int qr,ll x) { upd_min_(1,1,n,ql,qr,x); }
    void upd_max(int ql,int qr,ll x) { upd_max_(1,1,n,ql,qr,x); }
    void upd_add(int ql,int qr,ll x) { upd_add_(1,1,n,ql,qr,x); }
    void upd_set(int ql,int qr,ll x) { upd_set_(1,1,n,ql,qr,x); }
    void upd_mod(int ql,int qr,ll x) { upd_mod_(1,1,n,ql,qr,x); }
    void upd_divide(int ql,int qr,ll x) { upd_divide_(1,1,n,ql,qr,x); }
    ll query_sum(int ql,int qr) { return query_sum_(1,1,n,ql,qr); }
    ll query_max(int ql,int qr) { return query_max_(1,1,n,ql,qr); }
    ll query_min(int ql,int qr) { return query_min_(1,1,n,ql,qr); }
};
int32_t main(){
    ios_base :: sync_with_stdio(0); cin.tie(0);
    int t=1;
    // cin>>t;
    while(t--){
      int n; cin>>n;
      int q; cin>>q;
      STBeats t(n);
      vector<ll> v(n+1);
      for(int i=1;i<=n;i++) cin>>v[i];
      t.build(v);
      while(q--){
        int type,l,r; cin>>type>>l>>r;
        if(type==1) {
          ll val; cin>>val;
          t.upd_divide(l,r,val);
        }else if(type==2){
          ll val; cin>>val;
          t.upd_set(l,r,val);
        }else cout<<t.query_sum(l,r)<<endl;
      }
    }
}

/*
 @ the bellow code is dedicated for range and range divide it is more faster divide then main struct
 cause it is dedicated only for make divide very faster
*/

struct STBeats_Light {
private:
    struct node {
        ll sum;
        ll min1;
        ll max1;
        ll lazy_add;
        node() : sum(0), min1(INF), max1(NINF), lazy_add(0) {}
    };
    int n;
    vector<node> tree;
    void pull(int v) {
        tree[v].sum = tree[2 * v].sum + tree[2 * v + 1].sum;
        tree[v].min1 = min(tree[2 * v].min1, tree[2 * v + 1].min1);
        tree[v].max1 = max(tree[2 * v].max1, tree[2 * v + 1].max1);
    }
    void apply_add(int v, int tl, int tr, ll x) {
        tree[v].sum += (tr - tl + 1) * x;
        tree[v].min1 += x;
        tree[v].max1 += x;
        tree[v].lazy_add += x;
    }
    void push(int v, int tl, int tr) {
        if (tree[v].lazy_add == 0) return;
        int mid = (tl + tr) >> 1;
        apply_add(2 * v, tl, mid, tree[v].lazy_add);
        apply_add(2 * v + 1, mid + 1, tr, tree[v].lazy_add);
        tree[v].lazy_add = 0;
    }
    void build_(int v, int tl, int tr, const vector<ll>& a) {
        // here write the build function from main STBeats
    }
    void upd_add_(int v, int tl, int tr, int ql, int qr, ll x) {
        if (qr < tl || tr < ql) return;
        if (ql <= tl && tr <= qr) {
            apply_add(v, tl, tr, x);
            return;
        }
        push(v, tl, tr);
        int mid = (tl + tr) >> 1;
        upd_add_(2 * v, tl, mid, ql, qr, x);
        upd_add_(2 * v + 1, mid + 1, tr, ql, qr, x);
        pull(v);
    }
    ll floor_div(ll a, ll b) {
        // here write the floor_div function from main STBeats
    }
    void upd_divide_(int v, int tl, int tr, int ql, int qr, ll x) {
        if (qr < tl || tr < ql) return;
        if (ql <= tl && tr <= qr) {
            ll new_min = floor_div(tree[v].min1, x);
            ll new_max = floor_div(tree[v].max1, x);
            ll delta_min = new_min - tree[v].min1;
            ll delta_max = new_max - tree[v].max1;
            if (delta_min == delta_max) {
                apply_add(v, tl, tr, delta_min);
                return;
            }
            if (tl == tr) {
                ll new_val = floor_div(tree[v].min1, x);
                tree[v].sum = tree[v].min1 = tree[v].max1 = new_val;
                return;
            }
            push(v, tl, tr);
            int mid = (tl + tr) >> 1;
            upd_divide_(2 * v, tl, mid, ql, qr, x);
            upd_divide_(2 * v + 1, mid + 1, tr, ql, qr, x);
            pull(v);
        }
    ll query_sum_(int v, int tl, int tr, int ql, int qr) {
        // here write the query_sum_ function from main STBeats
    }
    ll query_min_(int v, int tl, int tr, int ql, int qr) {
        // here write the query_min_ function from main STBeats
    }

public:
    STBeats_Light(int n_val) : n(n_val) { tree.resize(4 * n + 4); }
    void build(const vector<ll>& a) { build_(1, 1, n, a); }
    void upd_add(int ql, int qr, ll x) { upd_add_(1, 1, n, ql, qr, x); }
    void upd_divide(int ql, int qr, ll x) { upd_divide_(1, 1, n, ql, qr, x); }
    ll query_sum(int ql, int qr) { return query_sum_(1, 1, n, ql, qr); }
    ll query_min(int ql, int qr) { return query_min_(1, 1, n, ql, qr); }
};
```

SEGTree Beats Bit and Gcd

Description:

1. **Bitwise:** Range AND/OR/Set using `tree[v].all_or` and `tree[v].all_and` to detect if update affects range.
2. **GCD:** Range GCD + Min/Max/Add.

```
#include"bits/stdc++.h"
using namespace std;
using ll=long long;
#define endl '\n'
const ll INF=1e18;
const ll NINF=-1e18;

struct STBeats_Bit {
private:
    struct node {
        ll sum;
        int len;
        ll all_and;
        ll all_or;
        ll max_val;
        ll min_val;
        ll lazy_set;

        node() : sum(0),len(0),all_and(~0LL),all_or(0LL),
            max_val(NINF),min_val(INF), lazy_set(INF) {}
    };
    int n;
    vector<node> tree;
    node merge(const node& left,const node& right) {
        node res;
        res.sum=left.sum+right.sum;
        res.len=left.len+right.len;
        res.all_and=left.all_and & right.all_and;
        res.all_or=left.all_or | right.all_or;
        res.max_val=max(left.max_val,right.max_val);
        res.min_val=min(left.min_val,right.min_val);
        res.lazy_set=INF;
        return res;
    }
    void apply_set(int v,ll x) {
        tree[v].sum=tree[v].len*x;
        tree[v].all_and=x;
        tree[v].all_or=x;
        tree[v].max_val=x;
        tree[v].min_val=x;
        tree[v].lazy_set=x;
    }
    void push_down(int v,int tl,int tr) {
        if(tl==tr || tree[v].lazy_set==INF) return;
        apply_set(2*v,tree[v].lazy_set);
        apply_set(2*v+1,tree[v].lazy_set);
        tree[v].lazy_set=INF;
    }
    void build_(int v,int tl,int tr,const vector<ll>& a) {
        if(tl==tr) {
            tree[v].len=1;
            tree[v].sum=a[tl];
            tree[v].all_and=a[tl];
            tree[v].all_or=a[tl];
            tree[v].max_val=a[tl];
            tree[v].min_val=a[tl];
        }else {
            int mid=(tl+tr)>>1;
            build_(2*v,tl,mid,a);
            build_(2*v+1,mid+1,tr,a);
            tree[v]=merge(tree[2*v],tree[2*v+1]);
        }
    }
    void upd_or_(int v,int tl,int tr,int ql, int qr,ll x) {
        push_down(v,tl,tr);
        if(qr<tl || tr<ql) return;
        if((tree[v].all_and & x)==x) return;
        if(tl==tr) {
            apply_set(v,tree[v].sum | x);
```

```cpp
        return;
    }
    int mid=(tl+tr)>>1;
    upd_or_(2*v,tl,mid,ql,qr,x);
    upd_or_(2*v+1,mid+1,tr,ql,qr,x);
    tree[v]=merge(tree[2*v],tree[2*v+1]);
}
void upd_and_(int v,int tl,int tr,int ql,int qr,ll x) {
    push_down(v,tl,tr);
    if(qr<tl || tr<ql) return;
    if((tree[v].all_or | x)==x) return;
    if(tl==tr) {
        apply_set(v,tree[v].sum & x);
        return;
    }
    int mid=(tl+tr)>>1;
    upd_and_(2*v,tl,mid,ql,qr,x);
    upd_and_(2*v+1,mid+1,tr,ql,qr,x);
    tree[v]=merge(tree[2*v],tree[2*v+1]);
}
void upd_set_(int v,int tl,int tr,int ql,int qr,ll x) {
    push_down(v,tl,tr);
    if(qr<tl || tr<ql) return;
    if(ql<=tl && tr<=qr) {
        apply_set(v,x);
        return;
    }
    int mid=(tl+tr)>>1;
    upd_set_(2*v,tl,mid,ql,qr,x);
    upd_set_(2*v+1,mid+1,tr,ql,qr,x);
    tree[v]=merge(tree[2*v],tree[2*v+1]);
}
ll query_sum_(int v,int tl,int tr,int ql,int qr) {
    if(qr<tl || tr<ql) return 0;
    push_down(v,tl,tr);
    if(ql<=tl && tr<=qr) return tree[v].sum;
    int mid=(tl+tr)>>1;
    return query_sum_(2*v,tl,mid,ql,qr) +
        query_sum_(2*v+1,mid+1,tr,ql,qr);
}
ll query_and_(int v,int tl,int tr,int ql,int qr) {
    if(qr<tl || tr<ql) return ~0LL;
    push_down(v,tl,tr);
    if(ql<=tl && tr<=qr) return tree[v].all_and;
    int mid=(tl+tr)>>1;
    return query_and_(2*v,tl,mid,ql,qr) &
        query_and_(2*v+1,mid+1,tr,ql,qr);
}
ll query_or_(int v,int tl,int tr,int ql, int qr) {
    if(qr<tl || tr<ql) return 0LL;
    push_down(v,tl,tr);
    if(ql<=tl && tr<=qr) return tree[v].all_or;
    int mid=(tl+tr)>>1;
    return query_or_(2*v,tl,mid,ql,qr) |
        query_or_(2*v+1,mid+1,tr,ql,qr);
}
ll query_max_(int v,int tl,int tr,int ql,int qr) {
    if(qr<tl || tr<ql) return NINF;
    push_down(v,tl,tr);
    if(ql<=tl && tr<=qr) return tree[v].max_val;
    int mid=(tl+tr)>>1;
    return max(query_max_(2*v,tl,mid,ql,qr) ,
        query_max_(2*v+1,mid+1,tr,ql,qr));
}
ll query_min_(int v,int tl,int tr,int ql,int qr) {
    if(qr<tl || tr<ql) return INF;
    push_down(v,tl,tr);
    if(ql<=tl && tr<=qr) return tree[v].min_val;
    int mid=(tl+tr)>>1;
    return min(query_min_(2*v,tl,mid,ql,qr) ,
        query_min_(2*v+1,mid+1,tr,ql,qr));
}

public:
    STBeats_Bit(int n) : n(n) { tree.resize(4*n+4); }
    void build(const vector<ll>& a) { build_(1,1,n,a); }
    void upd_or(int ql,int qr,ll x) { upd_or_(1,1,n,ql,qr,x); }
    void upd_and(int ql,int qr,ll x) { upd_and_(1,1,n,ql,qr,x); }
    void upd_set(int ql,int qr,ll x) { upd_set_(1,1,n,ql,qr,x); }
    ll query_sum(int ql,int qr) { return query_sum_(1,1,n,ql,qr); }
    ll query_and(int ql,int qr) { return query_and_(1,1,n,ql,qr); }
    ll query_or(int ql,int qr) { return query_or_(1,1,n,ql,qr); }
    ll query_max(int ql,int qr) { return query_max_(1,1,n,ql,qr); }
    ll query_min(int ql,int qr) { return query_min_(1,1,n,ql,qr); }
};
int32_t main() {
    ios_base :: sync_with_stdio(0); cin.tie(0);

    int n,q; cin>>n>>q;
    vector<ll> a(n+1);
    for(int i=1;i<=n;i++) cin>>a[i];
    STBeats_Bit t(n);
    t.build(a);
}
```

```cpp
/*
this code is for gcd and multiple update like cmax cmin add set
*/

#include<bits/stdc++.h>

using namespace std;

const long long MX = 1e18;

struct node {
    long long max, max2, min, min2, sum, gcd, add = 0, set = 0, updmin = 0, updmax = 0;
    int cntmax, cntmin;
    node() {}
    node(long long x) {
        sum = max = min = x, cntmax = cntmin = 1;
        gcd = 0;
        max2 = -MX, min2 = MX;
    }
};

vector<node> t;
vector<long long> a;

void merge(node& res, node& a, node& b) {
    // max
    res.max = max(a.max, b.max);
    res.max2 = -MX;
    res.cntmax = 0;
    if (a.max == res.max) {
        res.cntmax += a.cntmax;
        res.max2 = max(res.max2, a.max2);
    } else {
        res.max2 = max(res.max2, a.max);
    }
    if (b.max == res.max) {
        res.cntmax += b.cntmax;
        res.max2 = max(res.max2, b.max2);
    } else {
        res.max2 = max(res.max2, b.max);
    }

    // min
    res.min = min(a.min, b.min);
    res.min2 = MX;
    res.cntmin = 0;
    if (a.min == res.min) {
        res.cntmin += a.cntmin;
        res.min2 = min(res.min2, a.min2);
    } else {
        res.min2 = min(res.min2, a.min);
    }
    if (b.min == res.min) {
        res.cntmin += b.cntmin;
        res.min2 = min(res.min2, b.min2);
    } else {
        res.min2 = min(res.min2, b.min);
    }

    //sum
    res.sum = a.sum + b.sum;

    //gcd
    res.gcd = __gcd(a.gcd, b.gcd);
    long long x = -1, y = -1;
    if (a.max2 != -MX && a.max2 != a.min) {
        x = a.max2;
    }
    if (b.max2 != -MX && b.max2 != b.min) {
        y = b.max2;
    }
    if (x != -1 && y != -1) {
        res.gcd = __gcd(res.gcd, abs(x - y));
    }
    for (long long z : {a.max, a.min, b.max, b.min}) {
        if (z == res.max) {
            continue;
        }
        if (z == res.min) {
            continue;
        }
        if (x != -1) {
            res.gcd = __gcd(res.gcd, abs(x - z));
        } else if (y != -1) {
            res.gcd = __gcd(res.gcd, abs(y - z));
        } else {
            x = z;
        }
    }
}

void push_add(int v, long long x) {
    if (t[v].set != 0) {
        t[v].set += x;
    } else {
        if (t[v].updmin != 0) {
            t[v].updmin += x;
        }
        if (t[v].updmax != 0) {
            t[v].updmax += x;
        }
        t[v].add += x;
    }
}

void push_max(int v, long long x) {
    if (t[v].set != 0) {
        t[v].set = min(t[v].set, x);
    } else if (t[v].updmin == 0 || x > t[v].updmin) {
        if (t[v].updmax == 0) {
            t[v].updmax = x;
        } else {
            t[v].updmax = min(t[v].updmax, x);
        }
    } else {
        t[v].set = x;
    }
}

void push_min(int v, long long x) {
```

```
    if (t[v].set != 0) {
        t[v].set = max(t[v].set, x);
    } else if (t[v].updmax == 0 || t[v].updmax > x) {
        if (t[v].updmin == 0) {
            t[v].updmin = x;
        } else {
            t[v].updmin = max(t[v].updmin, x);
        }
    } else {
        t[v].set = x;
    }
}

void push(int v, int l, int r) {
    if (t[v].set != 0) {
        if (l + 1 != r) {
            t[v * 2 + 1].set = t[v * 2 + 2].set = t[v].set;
        }
        t[v].max = t[v].min = t[v].set;
        t[v].cntmax = t[v].cntmin = r - l;
        t[v].sum = t[v].set * (long long) (r - l);
        t[v].add = t[v].set = t[v].gcd = t[v].updmin = t[v].updmax = 0;
        t[v].max2 = -MX, t[v].min2 = MX;
    }
    if (t[v].add != 0) {
        if (l + 1 != r) {
            push_add(v * 2 + 1, t[v].add);
            push_add(v * 2 + 2, t[v].add);
        }
        t[v].max += t[v].add;
        t[v].min += t[v].add;
        if (t[v].max2 != -MX) {
            t[v].max2 += t[v].add;
        }
        if (t[v].min2 != MX) {
            t[v].min2 += t[v].add;
        }
        t[v].sum += t[v].add * (long long) (r - l);
        t[v].add = 0;
    }
    if (t[v].updmax != 0) {
        if (l + 1 != r) {
            push_max(v * 2 + 1, t[v].updmax);
            push_max(v * 2 + 2, t[v].updmax);
        }
        if (t[v].max == t[v].min) {
            if (t[v].updmax < t[v].max) {
                t[v].sum = t[v].updmax * (long long) (r - l);
                t[v].max = t[v].min = t[v].updmax;
            }
        } else {
            if (t[v].updmax < t[v].max) {
                t[v].sum -= (t[v].max - t[v].updmax) * (long long) t[v].cntmax;
                if (t[v].max == t[v].min2) {
                    t[v].min2 = t[v].updmax;
                }
                t[v].max = t[v].updmax;
            }
        }
        t[v].updmax = 0;
    }
    if (t[v].updmin != 0) {
        if (l + 1 != r) {
            push_min(v * 2 + 1, t[v].updmin);
            push_min(v * 2 + 2, t[v].updmin);
        }
        if (t[v].max == t[v].min) {
            if (t[v].updmin > t[v].min) {
                t[v].sum = t[v].updmin * (long long) (r - l);
                t[v].max = t[v].min = t[v].updmin;
            }
        } else {
            if (t[v].updmin > t[v].min) {
                t[v].sum += (t[v].updmin - t[v].min) * (long long) t[v].cntmin;
                if (t[v].min == t[v].max2) {
                    t[v].max2 = t[v].updmin;
                }
                t[v].min = t[v].updmin;
            }
        }
        t[v].updmin = 0;
    }
}

void build(int v, int l, int r) {
    if (l + 1 == r) {
        t[v] = node(a[l]);
        return;
    }
    int m = (l + r) / 2;
    build(v * 2 + 1, l, m);
    build(v * 2 + 2, m, r);
    merge(t[v], t[v * 2 + 1], t[v * 2 + 2]);
}

void updatemin(int v, int l, int r, int l1, int r1, long long x) {
    push(v, l, r);
    if (l1 >= r || l >= r1 || t[v].max <= x) return;
    if (l1 <= l && r <= r1 && t[v].max2 < x) {
        t[v].updmax = x;
        push(v, l, r);
        return;
    }
    int m = (l + r) / 2;
    updatemin(v * 2 + 1, l, m, l1, r1, x);
    updatemin(v * 2 + 2, m, r, l1, r1, x);
    merge(t[v], t[v * 2 + 1], t[v * 2 + 2]);
}

void updatemax(int v, int l, int r, int l1, int r1, long long x) {
    push(v, l, r);
    if (l1 >= r || l >= r1 || t[v].min >= x) return;
    if (l1 <= l && r <= r1 && t[v].min2 > x) {
        t[v].updmin = x;
        push(v, l, r);
        return;
    }
    int m = (l + r) / 2;
    updatemax(v * 2 + 1, l, m, l1, r1, x);
    updatemax(v * 2 + 2, m, r, l1, r1, x);
    merge(t[v], t[v * 2 + 1], t[v * 2 + 2]);
}

void updateset(int v, int l, int r, int l1, int r1, long long x) {
    push(v, l, r);
    if (l1 >= r || l >= r1) return;
    if (l1 <= l && r <= r1) {
        t[v].set = x;
        push(v, l, r);
        return;
    }
    int m = (l + r) / 2;
    updateset(v * 2 + 1, l, m, l1, r1, x);
    updateset(v * 2 + 2, m, r, l1, r1, x);
    merge(t[v], t[v * 2 + 1], t[v * 2 + 2]);
}

void updateadd(int v, int l, int r, int l1, int r1, long long x) {
    push(v, l, r);
    if (l1 >= r || l >= r1) return;
    if (l1 <= l && r <= r1) {
        t[v].add = x;
        push(v, l, r);
        return;
    }
    int m = (l + r) / 2;
    updateadd(v * 2 + 1, l, m, l1, r1, x);
    updateadd(v * 2 + 2, m, r, l1, r1, x);
    merge(t[v], t[v * 2 + 1], t[v * 2 + 2]);
}

long long getsum(int v, int l, int r, int l1, int r1) {
    push(v, l, r);
    if (l1 >= r || l >= r1) return 0ll;
    if (l1 <= l && r <= r1) return t[v].sum;
    int m = (l + r) / 2;
    return getsum(v * 2 + 1, l, m, l1, r1) + getsum(v * 2 + 2, m, r, l1, r1);
}

long long getmin(int v, int l, int r, int l1, int r1) {
    push(v, l, r);
    if (l1 >= r || l >= r1) return MX;
    if (l1 <= l && r <= r1) return t[v].min;
    int m = (l + r) / 2;
    return min(getmin(v * 2 + 1, l, m, l1, r1), getmin(v * 2 + 2, m, r, l1, r1));
}

long long getmax(int v, int l, int r, int l1, int r1) {
    push(v, l, r);
    if (l1 >= r || l >= r1) return -MX;
    if (l1 <= l && r <= r1) return t[v].max;
    int m = (l + r) / 2;
    return max(getmax(v * 2 + 1, l, m, l1, r1), getmax(v * 2 + 2, m, r, l1, r1));
}

long long getgcd(int v, int l, int r, int l1, int r1) {
    push(v, l, r);
    if (l1 >= r || l >= r1) return 0ll;
    if (l1 <= l && r <= r1) {
        long long res = __gcd(t[v].max, t[v].min);
        if (t[v].max2 != t[v].min && t[v].max2 != -MX) {
            res = __gcd(res, t[v].gcd);
            res = __gcd(res, t[v].max2);
        }
        return res;
    }
    int m = (l + r) / 2;
    return __gcd(getgcd(v * 2 + 1, l, m, l1, r1), getgcd(v * 2 + 2, m, r, l1, r1));
}
```

SEGTree with Hashing

```
const ll p=137; const ll N=2e5+10; // check range
const pair<ll,ll> mod={127657753,987654319};

ll powerr(ll a,ll b,ll mod){
  ll r=1;
  while(b){
    if(b%2) r=((r%mod) *(a%mod))%mod;
    a=((a%mod)*(a%mod))%mod;
    b/=2;
  }
  return r;
}
ll add(ll a,ll b,ll mod){return ((a%mod)+(b%mod)+mod)%mod;}
ll substract(ll a,ll b,ll mod){return ((a%mod)-(b%mod)+mod)%mod;}
ll mult(ll a,ll b,ll mod) {return ((a%mod)*(b%mod))%mod;}
ll fn(char ch){if(islower(ch)) return ch-'a'+1;if(isupper(ch)) return ch-'A'+1; return ch-'0'+1;}
// ll fn(ll a[i]) return a[i]; //for integer hash

pair<ll,ll> pw[N+10],inv[N+10],inv_p_minus1;
void precal(){
```

```
  pw[0].F=pw[0].S=1;
  for(int i=1;i<N;i++){
    pw[i].F=mult(pw[i-1].F,p,mod.F);
    pw[i].S=mult(pw[i-1].S,p,mod.S);
  }
  ll pw_inv1=powerr(p,mod.F-2,mod.F);
  ll pw_inv2=powerr(p,mod.S-2,mod.S);
  inv[0].F=inv[0].S=1;
  for(int i=1;i<N;i++){
    inv[i].F=mult(inv[i-1].F,pw_inv1,mod.F);
    inv[i].S=mult(inv[i-1].S,pw_inv2,mod.S);
  }
    inv_p_minus1 = {
        powerr(p-1, mod.F-2, mod.F),
        powerr(p-1, mod.S-2, mod.S)
    };
}
struct hashing {
  vector<pair<ll,ll>> t;
  vector<char>lazy; // lazy of integer for integer hash
  string s; // integer hash make vector<ll> a
  hashing(){}
  hashing(string _s){
    s=_s;
    ll n=s.size();
    t.resize(n*4);
    lazy.resize(n*4,'?');
  }
  inline void push(int node,int l,int r){
    if(lazy[node]=='?') return;
    ll len=(r-l+1);
    ll sum1 = mult(mult(substract(pw[len].F, 1, mod.F), inv_p_minus1.F, mod.F), pw[l].F, mod.F);
    ll sum2 = mult(mult(substract(pw[len].S, 1, mod.S), inv_p_minus1.S, mod.S), pw[l].S, mod.S);

    t[node].F = mult(sum1, fn(lazy[node]), mod.F);
    t[node].S = mult(sum2, fn(lazy[node]), mod.S);
    if(l!=r){
        lazy[node*2]=lazy[node*2+1]=lazy[node];
    }
    lazy[node]='?';
  }
  inline void here(int node){
      t[node].F=add(t[node*2].F,t[node*2+1].F,mod.F);
      t[node].S=add(t[node*2].S,t[node*2+1].S,mod.S);
  }
  void build(int node,int l,int r){
    if(l==r){
        t[node].F=mult(pw[l].F,fn(s[l]),mod.F);
        t[node].S=mult(pw[l].S,fn(s[l]),mod.S);
        return;
    }
    ll mid=(l+r)>>1;
    build(node*2,l,mid);
    build(node*2+1,mid+1,r);
    here(node);
  }
  void upd(int node,int l,int r,int i,int j, char value){
    push(node,l,r);
    if(l>j || r<i) return;
    if(i<=l && r<=j){
        lazy[node]=value;
        push(node,l,r);
        return;
    }
    ll mid=(l+r)>>1;
    upd(node*2,l,mid,i,j,value);
    upd(node*2+1,mid+1,r,i,j,value);
    here(node);
  }
  pair<ll,ll> query(int node,int l,int r,int i,int j){
    push(node,l,r);
    if(l>j || r<i) return {0,0}; /// check here
    if(i<=l && r<=j) return t[node];
    ll mid=(l+r)>>1;
    pair<ll,ll> x=query(node*2,l,mid,i,j);
    pair<ll,ll> y=query(node*2+1,mid+1,r,i,j);
    return {add(x.F,y.F,mod.F),add(x.S,y.S,mod.S)};
  }
  pair<ll,ll> get_hash(int l,int r,int n){
    pair<ll,ll> ck=query(1,0,n-1,l,r);
    ck.F=mult(ck.F,inv[l].F,mod.F);
    ck.S=mult(ck.S,inv[l].S,mod.S);
    return ck;
  }
}a;
int main(){
  precal();
  ll n,m,x; cin>>n>>m>>x;
  ll q=m+x;
  string s; cin>>s;
  a = hashing(s);
  a.build(1,0,n-1);
  while(q--){
    ll i; cin>>i;
    if(i==1){
      ll l,r; char c; cin>>l>>r>>c; l--,r--;
      a.upd(1,0,n-1,l,r,c);
    }else{
      ll l,r,d; cin>>l>>r>>d;
      --l,--r;
      if(d==(r-l+1) || a.get_hash(l,r-d,n)==a.get_hash(l+d,r,n))
        cout<<"YES"<<endl;
      else cout<<"NO"<<endl;
    }
  }
}
```

Number Theory

Modular Arithmetic

Description: Struct for auto-modular arithmetic. Supports +, -, *, /, pow, inv. Includes Factorial and nCr/nPr precomputation.

Time: $\mathcal{O}(N)$ precomputation, $\mathcal{O}(1)$ queries.

```
const int MOD = 1000000007;
template <ll M>
struct modint
{
  static ll _pow(ll n, ll k)
  {
    ll r = 1;
    for (; k > 0; k >>= 1, n = (n * n) % M)
      if (k & 1)
        r = (r * n) % M;
    return r;
  }
  ll v;
  modint(ll n = 0) : v(n % M) { v += (M & (0 - (v < 0))); }
  friend string to_string(const modint n) { return to_string(n.v); }
  friend istream &operator>>(istream &i, modint &n) { return i >> n.v; }
  friend ostream &operator<<(ostream &o, const modint n) { return o << n.v; }
  template <typename T>
  explicit operator T() { return T(v); }
  friend bool operator==(const modint n, const modint m) { return n.v == m.v; }
  friend bool operator!=(const modint n, const modint m) { return n.v != m.v; }
  friend bool operator<(const modint n, const modint m) { return n.v < m.v; }
  friend bool operator<=(const modint n, const modint m) { return n.v <= m.v; }
  friend bool operator>(const modint n, const modint m) { return n.v > m.v; }
  friend bool operator>=(const modint n, const modint m) { return n.v >= m.v; }
  modint &operator+=(const modint n)
  {
    v += n.v;
    v -= (M & (0 - (v >= M)));
    return *this;
  }
  modint &operator-=(const modint n)
  {
    v -= n.v;
    v += (M & (0 - (v < 0)));
    return *this;
  }
  modint &operator*=(const modint n)
  {
    v = (v * n.v) % M;
    return *this;
  }
  modint &operator/=(const modint n)
  {
    v = (v * _pow(n.v, M - 2)) % M;
    return *this;
  }
  friend modint operator+(const modint n, const modint m) { return modint(n) += m; }
  friend modint operator-(const modint n, const modint m) { return modint(n) -= m; }
  friend modint operator*(const modint n, const modint m) { return modint(n) *= m; }
  friend modint operator/(const modint n, const modint m) { return modint(n) /= m; }
  modint &operator++() { return *this += 1; }
  modint &operator--() { return *this -= 1; }
  modint operator++(int)
  {
    modint t = *this;
    return *this += 1, t;
  }
  modint operator--(int)
  {
    modint t = *this;
    return *this -= 1, t;
  }
  modint operator+() { return *this; }
  modint operator-() { return modint(0) -= *this; }
  modint pow(const ll k) const
  {
    return k < 0 ? _pow(v, M - 1 - (-k % (M - 1))) : _pow(v, k);
  }
  modint inv() const { return _pow(v, M - 2); }
};
using mint = modint<int(MOD)>;
void precompute()
{
  fact[0] = 1;
  for (int i = 1; i < MAXN; ++i)
  {
    fact[i] = fact[i - 1] * i % MOD;
  }
  mint cur = fact[MAXN - 1];
  cur = cur.inv();
  inv_fact[MAXN - 1] = cur.v;
  for (int i = MAXN - 2; i >= 0; --i)
  {
    inv_fact[i] = inv_fact[i + 1] * (i + 1) % MOD;
  }
}
long long nCr(int n, int r)
{
  if (r < 0 || r > n)
    return 0;
```

```cpp
  return fact[n] * inv_fact[r] % MOD *
      ↪ inv_fact[n - r] % MOD;
}
long long nPr(int n, int r)
{
  if (r < 0 || r > n)
    return 0;
  return fact[n] * inv_fact[n - r] % MOD;
}
```

Sieve & Primes

Description: Linear Sieve (spf), Segmented Sieve, Segmented Factorization, $\phi(n)$ (Euler Totient), Factorization.
Time: Sieve $\mathcal{O}(N)$, Factorize $\mathcal{O}(\log N)$ (with spf) or $\mathcal{O}(\sqrt{N})$.

```cpp
struct NumberTheory
{
  static ll power(ll x, ll n)
  {
    ll res = 1;
    while (n > 0)
    {
      if (n & 1)
        res *= x;
      x *= x;
      n >>= 1;
    }
    return res;
  }
  vector<ll> primes;
  vector<int> spf;
  void sieve(ll n)
  { // O(n)
    spf.assign(n + 1, 0);
    for (int i = 2; i <= n; ++i)
    {
      if (!spf[i])
      {
        spf[i] = i;
        primes.PB(i);
      }
      for (auto j : primes)
      {
        ll prime = j;
        ll composite_num = 1LL * i * prime;
        if (composite_num > n)
          break;
        spf[composite_num] = prime;
        if (prime == spf[i])
          break;
      }
    }
  }
  vector<ll> segmentedSieve(ll L, ll R)
  {
    vector<bool> mark(R - L + 1, true);
    if (L == 1)
      mark[0] = false;
    for (auto p : primes)
    {
      if (1LL * p * p > R)
        break;
      ll base = max(p * p, ((L + p - 1) / p)
          ↪ * p);
      for (ll j = base; j <= R; j += p)
        mark[j - L] = false;
    }
    vector<ll> seg;
    for (ll i = 0; i <= R - L; i++)
      if (mark[i])
        seg.push_back(L + i);
    return seg;
  }
  vector<vector<ll>> segment_factor;
  void segment_fact(ll L, ll R)
  {
    segment_factor.assign(R - L + 1, vector<
        ↪ ll>());
    vector<ll> range_primes(R - L + 1);
    for (ll i = 0; i <= R - L; i++)
      range_primes[i] = L + i;
    for (auto p : primes)
    {
      if (1LL * p * p > R)
        break;
      ll base = p * ((L + p - 1) / p);

      for (ll j = base; j <= R; j += p)
      {
        ll index = j - L;
        while (!(range_primes[index] % p))
        {
          segment_factor[index].PB(p);
          range_primes[index] /= p;
        }
      }
    }
    for (ll i = 0; i <= R - L; i++)
    {
      if (range_primes[i] <= 1)
        continue;

      segment_factor[i].PB(range_primes[i]);
    }
  }
  vector<ll> factorize(ll n)
  {
    vector<ll> f;
    for (auto p : primes)
    {
      if (1LL * p * p > n)
        break;
      while (n % p == 0)
      {
        f.push_back(p);
        n /= p;
      }
    }
    if (n > 1)
      f.push_back(n);
    return f;
  }
  ll phi(ll n)
  {
    ll res = n;
    for (auto p : primes)
    {
      if (1LL * p * p > n)
        break;
      if (n % p == 0)
      {
        while (n % p == 0)
          n /= p;
        res -= res / p;
      }
    }
    if (n > 1)
      res -= res / n;
    return res;
  }
  ll phi2(ll n)
  {
    vector<ll> v = factorize(n);
    map<ll, ll> mp;
    ll res = 1;
    for (int i = 0; i < v.size(); ++i)
    {
      ll p = v[i], exp = 0;
      while (i < v.size() && v[i] == p)
      {
        exp++;
        i++;
      }
      i--;
      res *= power(p, exp - 1) * (p - 1);
    }
    return res;
  }
  static ll xorUpto(ll n)
  {
    ll x = n % 4;
    if (x == 0)
      return n;
    if (x == 1)
      return 1;
    if (x == 2)
      return n + 1;
    return 0;
  }
  static ll nCr(ll n, ll r)
  {
    if (r > n)
      return 0;
    r = min(r, n - r);
    ll res = 1;
    for (ll i = 1; i <= r; i++)
    {
      res = res * (n - i + 1) / i;
    }
    return res;
  }
} P;
```

Pollard Rho Miller Rabin

Description: Deterministic Miller-Rabin primality test (up to 10^{18}) and Pollard's Rho factorization. Requires `__int128` for modular multiplication to avoid overflow.
Time: Primality $\mathcal{O}(k \log^3 N)$, Factorization $\mathcal{O}(N^{1/4})$.

```cpp
// this is the topic to find prime fact of a
    ↪ big number
using ll = unsigned long long;
mt19937_64 rng(chrono::steady_clock::now().
    ↪ time_since_epoch().count());
ll rand(ll n) { return rng() % (n - 2) + 1;
    ↪ }
ll modMul(ll a,ll b,ll mod) {
    return (__int128)a*b%mod;
}
ll modPower(ll base,ll exp,ll mod) {
    ll res=1;
    base%=mod;
    while(exp>0) {
        if(exp%2==1) res=modMul(res,base,mod
            ↪ );
        base=modMul(base,base,mod);
        exp/=2;
    }
    return res;
}
ll gcd(ll a,ll b) {
    while(b) {
        a%=b;
        swap(a,b);
    }
    return a;
}
const int MAX_SIEVE=1000001;
vector<int> spf(MAX_SIEVE);
void init_sieve() {
    vector<int> primes;
    for(int i=2;i<MAX_SIEVE;++i) {
        if(!spf[i]) {
            spf[i]=i;
            primes.PB(i);
        }
        for(int p:primes) {
            if(i*(ll)p>=MAX_SIEVE) break;
            spf[i*p]=p;
            if(!(i%p)) break;
        }
    }
}
bool MillerRabin(ll n,ll a,ll d,int s) {
    ll x=modPower(a,d,n);
    if(x==1 || x==n-1) return true;
    for(int r=1;r<s;r++) {
        x=modMul(x,x,n);
        if(x==1) return false;
        if(x==n-1) return true;
    }
    return false;
}
bool isPrime(ll n) {
    if(n<=1) return false;
    if(n<MAX_SIEVE) return spf[n]==n;
    if(n==2 || n==3) return true;
    if(!(n%2)) return false;
    ll d=n-1;
    int s=0;
    while(!(d%2)) {
        d/=2;
        s++;
    }
    vector<ll> witnesses
        ↪ ={2,3,5,7,11,13,17,19,23,29,31,37};
        ↪
    for(ll a:witnesses) {
```

```
        if(n==a) return true;
        if(!(MillerRabin(n,a,d,s))) return false;
    }
    return true;
}
ll pollard_rho(ll n) {
    auto f =[&](ll x,ll c) {
        return (modMul(x,x,n)+c)%n;
    };
    ll c=rand(n);
    ll tortoise=2,hare=2,d=1;
    ll product=1;
    const int BATCH_SIZE=128;
    int count=0;
    while(1) {
        tortoise=f(tortoise,c);
        hare=f(f(hare,c),c);
        if(tortoise==hare) {
            c=rand(n);
            tortoise=2; hare=2; product=1; count=0;
            continue;
        }
        ll prev_product=product,diff;
        if(tortoise>hare) diff=tortoise-hare ;
        else diff=hare-tortoise;
        product=modMul(product,diff,n);
        if(!product) {
            d=gcd(prev_product,n);
            if(d==1) d=gcd(diff,n);
            break;
        }
        count++;
        if(count==BATCH_SIZE) {
            d=gcd(product,n);
            if(d>1) break;
            count=0;
            product=1;
        }
    }
    if(d==n || d==1) return pollard_rho(n);
    return d;
}
void factorize(ll n,vector<ll>& primeFactors ) {
    if(n<=1) return;
    while(!(n%2)) {
        primeFactors.PB(2);
        n/=2;
    }
    if(n==1) return;
    while(n>1 && n<MAX_SIEVE) {
        primeFactors.PB(spf[n]);
        n/=spf[n];
    }
    if(n==1) return;
    if(isPrime(n)) {
        primeFactors.PB(n);
        return;
    }
    ll d=pollard_rho(n);
    factorize(d,primeFactors);
    factorize(n/d,primeFactors);
}
int32_t main() {
    init_sieve(); // run it before testcase
    ll n; cin>>n;
    vector<ll> ans;
    factorize(n,ans);
}
```

Mobius Function

Description: Linear Sieve to compute $\mu(i)$ and $\phi(i)$. $h[i]$ stores helper values for LCM sums.
Time: $\mathcal{O}(N)$.

```
const int MX=1000001;
vector<int> mu(MX);
vector<int> phi(MX);
vector<int> spf(MX);
vector<ll> h(MX,0); // for LCM
vector<int> primes;
void mobius_sieve(){
  mu[1]=1; h[1]=1;
  for(int i=2;i<MX;i++){
    if(!spf[i]){
        spf[i]=i;
        mu[i]=-1;
        phi[i]=i-1;
        h[i]=(1-i+MOD);
        primes.PB(i);
    }
    for(int p:primes){
        if(1LL*i*p>=MX) break;
        spf[i*p]=p;
        if(!(i%p)){
          h[i*p]=h[i];
          phi[i*p]=phi[i]*p;
          mu[i*p]=0;
          break;
        }else {
          mu[i*p]=-mu[i];
          phi[i*p]=phi[i]*(p-1);
          h[i*p]=(h[i]*h[p])%MOD;
        }
    }
  }
}
```

Mobius Inversion Formulas

Description:
1. count(n,k): Pairs with $\gcd(i,j) = k$. Uses $\sum_{d=1}^{\lfloor n/k \rfloor} \mu(d) \lfloor \frac{n}{kd} \rfloor^2$.
2. count(n): Sum of $\gcd(i,j)$ for $1 \le i,j \le n$.
3. solve_lcm: Count subsequences with LCM $= k$.
4. primitive: Count primitive strings.

```
// count gcd(i,j)==1 hard
// but count of gcd(i,j)%k==0 is easy cause i%k==0 and j%k==0
// that is N/k this much value can be divide by k and pairs are (N/k)*(N/k)
ll count(int n, int k)
{ // count gcd(i,j)==k i,j<=n
  n /= k;
  if (!n)
    return 0;
  ll ans = 0;
  for (int i = 1; i <= n; i++)
  { // this will find in O(n)
    ll g_i = (n / i) * (n / i);
    ans += 1LL * mu[i] * g_i;
  }
  return ans;
}
ll count_faster(int n, int k)
{ // this will find in O(sqrt n)
  n /= k;
  if (!n)
    return 0;
  ll ans = 0;
  for (int l = 1; l <= n;)
  {
    int val = n / l;
    int r = n / val;
    ll g_val = 1LL * val * val;
    ll mu_sum = mu_pre[r] - mu_pre[l - 1];
    ans += mu_sum * g_val;
    l = r + 1;
  }
  return ans;
}
ll count(ll n)
{
  ll ans = 0;
  for (int i = 1; i <= n;)
  {
    ll val = n / i;
    if (!val)
      break;
    ll r = n / val;
    ll g_val = (val * (val - 1)) / 2;
    ans += g_val * (pre_phi[r] - pre_phi[i - 1]);
    i = r + 1;
  }
  return ans;
}
void solve_lcm()
{ // ans for subsequence LCM=k;
  mobius_sieve();
  pow2[0] = 1;
  for (int i = 1; i < mx; i++)
    pow2[i] = pow2[i - 1] * 2;
  int n;
  cin >> n;
  map<int, int> freq;
  for (int i = 1; i <= n; i++)
  {
    int x;
    cin >> x;
    freq[x]++;
  }
  // now calculating the easy g[k] that is c[k]= count of numbers in A that divides k
  vector<int> c(mx, 0);
  for (auto const &[val, count] : freq)
  {
    for (int k = val; k < mx; k += val)
      c[k] += count;
  }
  vector<mi> g(mx);
  for (int k = 1; k < mx; k++)
  {
    g[k] = pow2[c[k]] - 1;
  }
  // f[n] = sum( g[d] * mu[n/d] )
  vector<mi> f(mx, 0);
  for (int d = 1; d < mx; d++)
  {
    // if(!g[d]) continue;
    for (int n = d; n < mx; n += d)
      f[n] += g[d] * mu[n / d];
  }
  // f[k] is the ans for subsequence LCM=k;
  int k;
  cin >> k;
  cout << f[k] << endl;
}
/*
*****Problem Statement: "Given N, and an alphabet of K letters,
find the number of primitive strings of length n for all n from 1 to N."
(A string is primitive if it's not a repetition of a smaller block,
e.g., "abcab" is primitive, but "ababab" is not).
*/
void solve_primitive_strings()
{
  int n = 100000, k = 26;
  mobius_sieve();
  vector<mi> g(n + 1);
  g[0] = 1;
  for (int i = 1; i <= n; i++)
    g[i] = g[i - 1] * k;
  vector<mi> f(mx, 0);
  for (int d = 1; d < mx; d++)
  {
    // if(!g[d]) continue;
    for (int n = d; n < mx; n += d)
      f[n] += g[d] * mu[n / d];
  }
  // cout << "Primitive strings of length 4 (K=26): " << f[4] << endl;
}
```

Mobius LCM Array

Description: Computes sum of LCM of all pairs in an array. Uses precomputed $h[i]$ from sieve.

```
  int n; cin>>n;
  int mx=0;
  vector<int> v(n+1);
  mi sum=0;
  for(int i=1;i<=n;i++) {
    cin>>v[i];
    mx=max(mx,v[i]);
    sum+=v[i];
  }
  vector<int>fre(mx+1,0);
  for(int i=1;i<=n;i++) fre[v[i]]++;
  vector<ll>mp(mx+1,0);
  for(int i=1;i<=mx;i++) {
    for(int j=i;j<=mx;j+=i) {
```

```
      ll k=j/i;
      mp[i]+=1LL*k*fre[j];
    }
  }
  mi ans=0;
  for(int i=1;i<=mx;i++) {
    mi term=mi(i)*mi(h[i])*mi(mp[i]);
    ans+=term;
  }
  cout<<ans<<endl; // all pair lcm sum
  cout<<mi(ans-sum)<<endl; // exclude i=j
  mi inv=mi((MOD+1)/2);
  cout<<mi(mi(ans-sum)*inv)<<endl; // all pair lcm i<j
```

Fast Prime Count

Description: Counts $\pi(n)$ (number of primes $\leq n$) in sub-linear time.
Time: $\mathcal{O}(N^{2/3})$.

```
const int N=3e5+9;
namespace pcf {
    #define MAXN 20000010
    #define MAX_PRIMES 2000010
    #define PHI_N 100000
    #define PHI_K 100
    int len=0; // number of prime gen by sieve
    int primes[MAX_PRIMES];
    int pref[MAXN]; // number of primes <=i
    int dp[PHI_N][PHI_K];
    bitset<MAXN> f;
    void sieve(int n) {
        f[1]=true;
        for(int i=4;i<=n;i+=2) f[i]=true;
        for(int i=3;i*i<=n;i+=2) {
            if(!f[i]) {
                for(int j=i*i;j<=n;j+=i<<1) f[j]=true;
            }
        }
        for(int i=1;i<=n;i++) {
            if(!f[i]) primes[len++]=i;
            pref[i]=len;
        }
    }
    void init() {
        sieve(MAXN-1);
        for(int n=0;n<PHI_N;n++) dp[n][0]=n;
        for(int k=1;k<PHI_K;k++) {
            for(int n=0;n<PHI_N;n++) {
                dp[n][k]=dp[n][k-1]-dp[n/primes[k-1]][k-1];
            }
        }
    }
    ll bro(ll n,int k) { // number of int <= n not div by first k primes
        if(n<PHI_N && k<PHI_K) return dp[n][k];
        if(k==1) return ((++n)>>1);
        if(primes[k-1]>=n) return 1;
        return bro(n,k-1)-bro(n/primes[k-1],k-1);
    }
    ll lehmer(ll n) { // runs under 0.2s for n=1e12
        if(n<MAXN) return pref[n];
        ll w,res=0;
        int b=sqrt(n),c=lehmer(cbrt(n)),a=lehmer(sqrt(b));b=lehmer(b);
        res=bro(n,a)+((1LL*(b+a-2)*(b-a+1))>>1);
        for(int i=a;i<b;i++) {
            w=n/primes[i];
            int lim=lehmer(sqrt(w)); res-=lehmer(w);
            if(i<=c) {
                for(int j=i;j<lim;j++) {
                    res+=j;
                    res-=lehmer(w/primes[j]);
                }
            }
        }
        return res;
    }
}
int32_t main() {
    pcf::init();
    ll n; cin>>n;
    cout<<pcf::lehmer(n)<<endl;
}
```

Extended EGCD

Description: Finds x, y such that $ax + by = \gcd(a, b)$. Returns gcd.
Time: $\mathcal{O}(\log(\min(a, b)))$.

```
ll extended_gcd(ll a, ll b, ll &x, ll &y) {
  if (b == 0) {
    x = 1, y = 0;
    return a;
  }
  ll x1, y1;
  ll d = extended_gcd(b, a % b, x1, y1);
  x = y1;
  y = x1 - y1 * (a / b);
  return d;
}
```

Catalan Number

Description: $C_n = \frac{1}{n+1}\binom{2n}{n}$. Counts valid parenthesis sequences, binary trees, polygon triangulations, etc.
Time: $\mathcal{O}(N)$.

```
ll dp[M], fac[2 * M];
void fact() {
  fac[0] = 1;
  for (int i = 1; i < 2 * M; i++)
    fac[i] = (fac[i - 1] * i) % mod;
}
void cal() {                          /// O(n*logn)
  dp[0] = dp[1] = 1; /// x = (2*x)!/((x+1)!*x!)
  for (int i = 2; i < M; i++)
    dp[i] =
    (fac[2*i]*bigmod((fac[i+1]*fac[i])%mod,
        mod-2,mod))%mod;
}
```

Custom Bitset (Dynamic)

```
// Compact, fast bitset wrapper using uint64_t blocks.
// - b : number of bits the bitset represents (logical length).
// - n : number of uint64_t words used = ceil(b / 64).
// - bits : underlying storage; bits[0] stores bits [0..63], bits[1] -> [64..127], etc.
//
// Notes:
// - Indexing and public methods use 0-based bit indices in range [0, b).
// - _clean() masks off unused high bits in the last word so count()/find_first() behave correctly.
// - left_shift/right_shift implement block+intra-block shifts using OR to accumulate results
//   (your implementation performs |= shifts; if you want pure shift (assignment) semantics,
//   you would need to zero the target before ORing).
struct Cool_Bitset {
  vector<uint64_t> bits;   // storage
  int64_t b, n;            // b = number of bits, n = number of 64-bit words
  // ctor: optional initial bit length
  Cool_Bitset(int64_t _b = 0) {
    init(_b);
  }
  // initialize to hold _b bits (all cleared)
  void init(int64_t _b) {
    b = _b;
    n = (b + 63) / 64;          // number of 64-bit words required
    bits.assign(n, 0);          // zero-initialize
  }
  // completely free storage
  void clear() {
    b = n = 0;
    bits.clear();
  }
  // reset contents to zero but keep size
  void reset() {
    bits.assign(n, 0);
  }
  // mask out unused high bits in the last word (if b is not a multiple of 64).
  // This ensures operations like count() and find_first() don't see garbage bits past 'b'.
  void _clean() {
    if (b != 64 * n) {
      // compute number of valid bits in last word and mask others off
      bits.back() &= (1ULL << (b - 64 * (n - 1))) - 1;
    }
  }
  // read bit at index (0-based). Returns 0/1.
  bool get(int64_t index) const {
    // no bounds check here for speed; caller should ensure 0 <= index < b
    return (bits[index / 64] >> (index % 64)) & 1ULL;
  }
  // write bit at index to 'value' (true => 1, false => 0)
  void set(int64_t index, bool value) {
    assert(0 <= index && index < b);
                // debug-only check
    int64_t word = index / 64;
    int shift = index % 64;
    // clear the target bit then set accordingly
    bits[word] &= ~(1ULL << shift);
    bits[word] |= (uint64_t(value) << shift);
  }

  // LEFT shift by 'shift' bits (logical shift). Implementation uses |= so it accumulates bits.
  // Complexity: O(n)
  void left_shift(int64_t shift) {
    int64_t div = shift / 64;    // whole-word shift
    int64_t mod = shift % 64;    // intra-word shift
    if (mod == 0) {
      // shift by whole words: move words upward
      for (int64_t i = n - 1; i >= div; i--)
        bits[i] |= bits[i - div];
      // note: words [0..div-1] are unchanged (ORed with 0)
      return;
    }
    // shift with both whole-word and bit offset
    for (int64_t i = n - 1; i >= div + 1; i--) {
      // combine higher-part and lower-part of source words
      bits[i] |= (bits[i - (div + 1)] >> (64 - mod)) | (bits[i - div] << mod);
    }
    // handle the boundary word (if any)
    if (div < n)
      bits[div] |= bits[0] << mod;
    _clean(); // ensure we didn't set bits past 'b'
  }
  // RIGHT shift by 'shift' bits (logical). Implementation uses |= so it
```

```
    accumulates bits.
// Complexity: O(n)
void right_shift(int64_t shift) {
  int64_t div = shift / 64;
  int64_t mod = shift % 64;
  if (mod == 0) {
    for (int64_t i = div; i < n; i++)
      bits[i - div] |= bits[i];
    return;
  }
  for (int64_t i = 0; i < n - (div + 1); i++)
    bits[i] |= (bits[i + (div + 1)] << (64 - mod)) | (bits[i + div] >> mod);
  if (div < n)
    bits[n - div - 1] |= bits[n - 1] >> mod;
  _clean();
}
// population count (number of set bits). Uses builtin popcountll on each word.
int64_t count() const {
  int64_t res = 0;
  for (int64_t i = 0; i < n; i++)
    res += __builtin_popcountll(bits[i]);
  return res;
}
// find index of first set bit (lowest index). Returns -1 if none.
// Complexity: O(n) in worst case, but fast because it scans word-by-word and uses ctz.
int64_t find_first() const {
  for (int64_t i = 0; i < n; i++)
    if (bits[i] != 0)
      return 64 * i + __builtin_ctzll(bits[i]); // ctz: count trailing zeros
  return -1;
}
// find next set bit strictly after x (i.e., search from x+1).
// Safety: original loop could read past 'b', so we added a guard that stops at 'b'.
// Returns -1 if none.
int64_t find_next(int64_t x) const {
  // first scan in the same word (from x+1 up to end of that word)
  int64_t start = x + 1;
  if (start < b) {
    int64_t end_same_word = min<int64_t>((x / 64) * 64 + 64, b ); // exclusive bound
    for (int64_t i = start; i < end_same_word; ++i) {
      if (get(i)) return i;
    }
  }

  // then scan entire following words
  for (int64_t i = x / 64 + 1; i < n; i++)
    if (bits[i] != 0)
      return 64 * i + __builtin_ctzll(bits[i]);

  return -1;
}
// in-place AND with another bitset (must be same size)
Cool_Bitset& operator&=(const Cool_Bitset &other) {
  assert(b == other.b);
  for (int64_t i = 0; i < n; i++)
    bits[i] &= other.bits[i];
  return *this;
}
// return new bitset = this & other
Cool_Bitset operator&(const Cool_Bitset & other) const {
  assert(b == other.b);
  Cool_Bitset res(b);
  for (int64_t i = 0; i < n; i++) res.bits[i] = bits[i] & other.bits[i];
  return res;
}
};
```

DP

LIS

Description: Finds the Longest Increasing Subsequence and reconstructs the solution. Returns `{length, indices, values}`. For non-decreasing (monotonic), change `lower_bound` to `upper_bound`.

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

```
tuple<int, vector<int>, vector<int>> LIS_with_path(const vector<int> &a)
{
  int n = a.size();
  vector<int> d;
  vector<int> d_idx;
  vector<int> parent(n, -1);
  vector<int> pos(n, -1);
  for (int i = 0; i < n; ++i) {
    int x = a[i];
    auto it = lower_bound(d.begin(), d.end(), x);
    int len = it - d.begin();
    if (it == d.end()) {
      d.push_back(x);
      d_idx.push_back(i);
    }
    else {
      *it = x;
      d_idx[len] = i;
    }
    pos[len] = i;
    if (len > 0)
      parent[i] = pos[len - 1];
  }
  int L = d.size();
  vector<int> indices(L);
  int cur = pos[L - 1];
  for (int k = L - 1; k >= 0; --k) {
    indices[k] = cur;
    cur = parent[cur];
  }
  vector<int> values(L);
  for (int i = 0; i < L; ++i)
    values[i] = a[indices[i]];
  return {L, indices, values};
}
```

Binary Optimization

Description: Solves Bounded Knapsack (limited count of items) by decomposing counts into powers of 2 $(1, 2, 4, \dots, rem)$. Turns $\mathcal{O}(W \cdot \text{count})$ into $\mathcal{O}(W \cdot \log(\text{count}))$.

Time: $\mathcal{O}(W \cdot \sum \log(\text{count}))$.

```
map<int, int> mp;
for (auto it : vec)
  mp[it]++;
vector<int> dp(n + 1, 1e9);
dp[0] = 0;
for (auto [w, cnt] : mp)
{
  int cur = 1;
  while (cnt > 0) {
    int use = min(cnt, cur);
    for (int i = n; i >= w * use; i--) {
      dp[i] = min(dp[i], dp[i - w * use] + use);
    }
    cnt -= use;
    cur *= 2;
  }
}
```

Geometry

2D Basics: Epsilon & Vectors

Description:

- **Precision:** Always use `long double` and `EPS` (10^{-9} or 10^{-12}) for comparisons. Never use `==` directly.
- **Dot Product** $(A \cdot B)$: $x_1x_2 + y_1y_2 = |A||B| \cos\theta$.
  - > 0: Angle is Acute ($< 90^\circ$).
  - $= 0$: Vectors are Perpendicular (90°).
  - < 0: Angle is Obtuse ($> 90^\circ$).
  - Used for: Projecting points onto lines, finding angles.
- **Cross Product** $(A \times B)$: $x_1y_2 - x_2y_1 = |A||B| \sin\theta$.
  - Represents the *signed area* of the parallelogram formed by A and B.
  - > 0: B is "Left" (CCW) of A.
  - < 0: B is "Right" (CW) of A.
  - Used for: Finding orientation, area, intersection checks.

```
using ld = long double;
const ld EPS = 1e-12L;
const ld PI = acosl(-1.0L);
inline int sgn(ld x) { if (x > EPS) return 1; if (x < -EPS) return -1; return 0; }
struct P {
    ld x, y;
    P(ld _x = 0, ld _y = 0) : x(_x), y(_y) {}
    P operator+(const P& o) const { return P(x+o.x, y+o.y); }
    P operator-(const P& o) const { return P(x-o.x, y-o.y); }
    P operator*(ld k) const { return P(x*k, y*k); }
    P operator/(ld k) const { return P(x/k, y/k); }
    bool operator==(const P& o) const {
        return sgn(x-o.x)==0 && sgn(y-o.y)==0; }
    bool operator!=(const P& o) const {
        return !(*this==o); }
};
// Operations
inline ld dot(const P& a, const P& b) {
    return a.x*b.x + a.y*b.y; }
inline ld cross(const P& a, const P& b) {
    return a.x*b.y - a.y*b.x; }
inline ld norm2(const P& a) { return dot(a,a); }
inline ld absP(const P& a) { return sqrtl(
    max((ld)0.0L, norm2(a))); }
// Rotate a by ang radians CCW
inline P rotate(const P& a, ld ang) { return
    P(a.x*cosl(ang) - a.y*sinl(ang), a.x*sinl(ang) + a.y*cosl(ang)); }
// Returns vector rotated 90 degrees CCW (-y, x)
inline P perp(const P& a) { return P(-a.y, a.x); }
// Returns angle in range [-PI, PI]
inline ld angleOf(const P& a) { return
    atan2l(a.y, a.x); }
```

Orientation (CCW)

Description: The most critical function in geometry.

- `ccw(a, b, c)` checks the turn made walking $a \to b \to c$.
- **Returns:** $+1$ (Left turn/CCW), -1 (Right turn/CW), 0 (Collinear).
- `onSegment`: Checks if c lies strictly on the segment ab. Condition: Must be collinear AND dot product checks if c is between a and b.

```
inline int ccw(const P& a, const P& b, const
    P& c) { return sgn(cross(b - a, c -
    a)); }

inline bool onSegment(const P& a, const P& b
    , const P& c) {
    // Must be collinear (cross product 0)
        and c must be between a and b
    if (ccw(a,b,c) != 0) return false;
    return sgn(dot(c - a, c - b)) <= 0;
}
```

Lines, Segments & Distances

Description:

- **Projection:** Finding the closest point on an infinite line. Derived from A + vector $\times$ (projection ratio).
- **Segment Intersection:** Uses the "Straddle Test". Segment CD intersects line AB if C and D are on opposite sides of AB (checked via `ccw`). Two segments intersect if they straddle each other.
- **Distances:**
  - `Pt-Line`: Simple perpendicular distance.
  - `Pt-Segment`: Closest point might be endpoints or the projection.

```
// Projection of p onto line AB
inline P projectPointLine(const P& a, const
```

```cpp
    P& b, const P& p) {
    P ap = p - a, ab = b - a;
    return a + ab * (dot(ap, ab) / norm2(ab));
}
// Reflection of p across line AB
inline P reflectPointLine(const P& a, const P& b, const P& p) {
    return projectPointLine(a,b,p)*2 - p;
}
// Checks if segments [a,b] and [c,d] intersect strictly or at endpoints
inline bool segmentsIntersect(const P& a, const P& b, const P& c, const P& d) {
    int d1 = ccw(a,b,c), d2 = ccw(a,b,d);
    int d3 = ccw(c,d,a), d4 = ccw(c,d,b);
    // General case: straddle test
    if (((d1 > 0 && d2 < 0) || (d1 < 0 && d2 > 0)) &&
        ((d3 > 0 && d4 < 0) || (d3 < 0 && d4 > 0))) return true;
    // Special case: collinear endpoints (touching)
    if (d1 == 0 && onSegment(a,b,c)) return true;
    if (d2 == 0 && onSegment(a,b,d)) return true;
    if (d3 == 0 && onSegment(c,d,a)) return true;
    if (d4 == 0 && onSegment(c,d,b)) return true;
    return false;
}
// Returns {exists, point}. Uses parametric eq P + tR = Q + uS
inline pair<bool, P> intersectLines(const P& p, const P& r, const P& q, const P& s) {
    ld rxs = cross(r, s);
    if (sgn(rxs) == 0) return {false, P()}; // Parallel
    ld t = cross(q - p, s) / rxs;
    return {true, p + r * t};
}
// Intersection of segments. Handles overlaps by returning average point.
inline pair<bool, P> intersectSegments(const P& a, const P& b, const P& c, const P& d) {
    auto res = intersectLines(a, b - a, c, d - c);
    if (!res.first) { // Parallel lines
        if (ccw(a, b, c) != 0) return {false, P()}; // Distinct parallel lines
        // Collinear overlap check
        vector<P> pts;
        if (onSegment(a,b,c)) pts.push_back(c);
        if (onSegment(a,b,d)) pts.push_back(d);
        if (onSegment(c,d,a)) pts.push_back(a);
        if (onSegment(c,d,b)) pts.push_back(b);
        if (pts.empty()) return {false, P()};
        P sum(0,0); for (auto &p: pts) sum = sum + p;
        return {true, sum / (ld)pts.size()};
    } else {
        P ip = res.second; // Check if intersection lies on both segments
        if (onSegment(a,b,ip) && onSegment(c,d,ip)) return {true, ip};
        return {false, P()};
    }
}
inline ld distancePointLine(const P& a, const P& b, const P& p) {
    return fabsl(cross(b - a, p - a)) / absP(b - a);
}
inline ld distancePointSegment(const P& a, const P& b, const P& p) {
    P ab = b - a, ap = p - a, bp = p - b;
    if (sgn(dot(ab, ap)) < 0) return absP(ap); // Closer to A
    if (sgn(dot(ab, bp)) > 0) return absP(bp); // Closer to B
    return distancePointLine(a, b, p);          // Perpendicular
}
```

Polygon Algorithms

Description:

- **Shoelace Formula:** Calculates area. Note that it returns *signed* area. Positive if vertices are CCW, negative if CW. Use `fabs` for magnitude.
- **Centroid:** Center of mass. Computed by weighting triangle centroids by their area.
- **Point in Polygon:** Uses Ray Casting. Draw a line from point to $+\infty$. If it crosses odd edges → inside. If even → outside.
- **Is Convex:** Checks if all consecutive edge pairs turn in the same direction.

```cpp
inline ld polygonAreaSigned(const vector<P>& poly) {
    ld A = 0;
    for (int i = 0; i < poly.size(); ++i)
        A += cross(poly[i], poly[(i+1)%poly.size()]);
    return A/2;
}
inline P polygonCentroid(const vector<P>& poly) {
    int n = poly.size();
    ld A2 = 0; P c(0,0);
    for (int i = 0; i < n; ++i) {
        ld cr = cross(poly[i], poly[(i+1)%n]);
        A2 += cr; c = c + (poly[i] + poly[(i+1)%n]) * cr;
    }
    return sgn(A2/2) == 0 ? P(0,0) : c / (3*A2);
}
// Returns: 0 = Outside, 1 = Inside, 2 = On Boundary
inline int pointInPolygon(const vector<P>& poly, const P& pt) {
    int n = poly.size(); bool inside = false;
    for (int i = 0; i < n; ++i) {
        P a = poly[i], b = poly[(i+1)%n];
        if (onSegment(a,b,pt)) return 2;
        // Ray casting logic: check intersections with horizontal ray to the right
        if ((sgn(a.y - pt.y) > 0) != (sgn(b.y - pt.y) > 0)) {
            ld xint = a.x + (b.x - a.x) * (pt.y - a.y) / (b.y - a.y);
            if (sgn(xint - pt.x) > 0) inside = !inside;
        }
    }
    return inside ? 1 : 0;
}
inline bool isConvex(const vector<P>& poly) {
    int n = poly.size(), initial = 0;
    if (n < 3) return false;
    for (int i = 0; i < n; ++i) {
        int c = ccw(poly[i], poly[(i+1)%n], poly[(i+2)%n]);
        if (c != 0) {
            if (initial == 0) initial = c;
            else if (initial != c) return false; // Direction changed (convexity broken)
        }
    }
    return true;
}
```

Convex Hull (Monotone Chain)

Description:

- Constructs the smallest convex polygon containing all points.
- **Method:** Sorts points by X (then Y). Builds "Lower Hull" (bottom curve) and "Upper Hull" (top curve) separately.
- **Key Logic:** While the last 3 points make a "Right Turn" (CW), the middle point is invalid (inside the hull), so pop it.
- **Time:** $\mathcal{O}(N \log N)$ (dominated by sorting).

```cpp
inline vector<P> convexHull(vector<P> pts) {
    int n = pts.size();
    if (n <= 1) return pts;
    sort(pts.begin(), pts.end(), point_cmp);
    vector<P> lower, upper;
    // Build lower hull
    for (int i = 0; i < n; ++i) {
        while (lower.size() >= 2 && ccw(lower[lower.size()-2], lower.back(), pts[i]) <= 0)
            lower.pop_back();
        lower.push_back(pts[i]);
    }
    // Build upper hull
    for (int i = n-1; i >= 0; --i) {
        while (upper.size() >= 2 && ccw(upper[upper.size()-2], upper.back(), pts[i]) <= 0)
            upper.pop_back();
        upper.push_back(pts[i]);
    }
    lower.pop_back(); upper.pop_back(); // Remove duplicate start/end points
    vector<P> ch = lower;
    ch.insert(ch.end(), upper.begin(), upper.end());
    return ch;
}
```

Circles

Description:

- **Circle-Circle Intersection:** Uses Law of Cosines to find the angle/distance to the intersection points. Returns 0, 1 (tangent), or 2 points.
- **Tangents:** Calculates points on a circle where a line from point P is tangent. Uses basic trig: $\sin(\alpha) = R/\text{dist}$.
- **Circumcircle:** Finds the circle passing through 3 points. Fails if points are collinear.

```cpp
struct Circle { P c; ld r; Circle(P _c=P(0,0), ld _r=0) : c(_c), r(_r) {} };

inline vector<P> circleCircleIntersection(const Circle& A, const Circle& B) {
    ld d = dist(A.c, B.c);
    // Cases: Too far apart || One inside other || Coincident
    if (d > A.r + B.r + EPS || d + min(A.r,B.r) + EPS < max(A.r,B.r) || d < EPS) return {};
    ld x = (d*d - B.r*B.r + A.r*A.r) / (2*d);
    ld y2 = max((ld)0.0L, A.r*A.r - x*x);
    P v = (B.c - A.c) / d, perpv = perp(v), p = A.c + v * x;
    if (sgn(y2) == 0) return {p}; // 1 intersection
    return {p + perpv * sqrtl(y2), p - perpv * sqrtl(y2)}; // 2 intersections
}
// Tangent points on C from external point p
inline vector<P> tangentsPointCircle(const P& p, const Circle& C) {
    ld d2 = dist2(p, C.c), r2 = C.r * C.r;
    if (d2 < r2 - EPS) return {}; // Point inside circle
    if (fabsl(d2 - r2) < EPS) return {p}; // Point on circle
    ld d = sqrtl(d2), l = r2 / d, h = sqrtl(max((ld)0.0L, r2 - l*l));
    P v = (p - C.c) / d, mid = C.c + v * l, perpv = perp(v);
    return {mid + perpv * h, mid - perpv * h};
}
// Circle passing through 3 points
inline Circle circumCircle(const P& a, const P& b, const P& c) {
    ld d = 2 * cross(b - a, c - a);
```

```cpp
    if (fabsl(d) < EPS) return Circle(P(0,0), -1); // Collinear
    ld A = norm2(a), B = norm2(b), Cc = norm2(c);
    P center((A*(b.y - c.y) + B*(c.y - a.y) + Cc*(a.y - b.y)) / d,
            (A*(c.x - b.x) + B*(a.x - c.x) + Cc*(b.x - a.x)) / d);
    return Circle(center, dist(center, a));
}
```

Closest Pair of Points

Description: Finds the minimum distance between any two points in a set.

- **Algorithm:** Divide & Conquer.
- **Logic:** 1. Sort points by X-coordinate. 2. Divide into left-/right halves. Recurse to find $d = \min(d_L, d_R)$. 3. **Merge Step:** The closest pair might span the dividing line. Gather points within distance d of the middle X-line into a "strip". 4. Sort strip by Y-coordinate. For each point, check neighbors in the strip. (Geometry guarantees we only need to check the next ≈ 7 points).
- **Time:** $\mathcal{O}(N \log N)$ (if we merge-sort by Y during recursion) or $\mathcal{O}(N \log^2 N)$ (if we sort strip explicitly). The code below uses `inplace_merge` for $\mathcal{O}(N \log N)$.

```cpp
// Auxiliary function for recursion
ld closestPairRec(vector<P>& pts, int l, int r, vector<P>& aux) {
    if (r - l <= 3) {
        ld best = numeric_limits<ld>::infinity();
        for (int i = l; i < r; ++i)
            for (int j = i+1; j < r; ++j)
                best = min(best, dist(pts[i], pts[j]));
        // Sort by Y for the merge step
        sort(pts.begin()+l, pts.begin()+r, [](const P& a, const P& b){ return a.y < b.y; });
        return best;
    }
    int m = (l + r) >> 1;
    ld midx = pts[m].x;
    ld d = min(closestPairRec(pts, l, m, aux), closestPairRec(pts, m, r, aux));

    // Merge both sorted halves by Y-coordinate
    inplace_merge(pts.begin()+l, pts.begin()+m, pts.begin()+r,
                  [](const P& a, const P& b) { return a.y < b.y; });

    // Create strip: only keep points within 'd' horizontal distance from midx
    int sz = 0;
    for (int i = l; i < r; ++i) {
        if (fabsl(pts[i].x - midx) < d + EPS) aux[sz++] = pts[i];
    }
    // Check points in strip against their neighbors (within vertical distance d)
    for (int i = 0; i < sz; ++i) {
        for (int j = i+1; j < sz && (aux[j].y - aux[i].y) < d + EPS; ++j) {
            d = min(d, dist(aux[i], aux[j]));
        }
    }
    return d;
}
inline ld closestPair(vector<P> pts) {
    sort(pts.begin(), pts.end(), point_cmp); // Sort by X initially
    vector<P> aux(pts.size());
    return closestPairRec(pts, 0, pts.size(), aux);
}
```

Advanced Distance & Circle Helpers

Description:

- **Seg-Seg Dist:** 1. If intersect $\to 0$. 2. Else: $\min(d(A,CD), d(B,CD), d(C,AB), d(D,AB))$.
- **Circle Centers (2 Pts):** Pts A, B, rad R:
  - $\text{dist}(A,B) > 2R \to$ impossible.
  - Centers on perp bisector of AB, dist $h = \sqrt{R^2 - (d/2)^2}$ from midpoint.

```cpp
// Minimum distance between two segments
inline ld distanceSegmentSegment(const P& a, const P& b, const P& c, const P& d) {
    if (segmentsIntersect(a,b,c,d)) return 0;
    ld ans = distancePointSegment(a,b,c);
    ans = min(ans, distancePointSegment(a,b,d));
    ans = min(ans, distancePointSegment(c,d,a));
    ans = min(ans, distancePointSegment(c,d,b));
    return ans;
}
// Find centers of circle(s) of radius r passing through a and b
inline vector<P> circleCentersFrom2Points(const P& a, const P& b, ld r) {
    ld d2 = dist2(a,b);
    ld d = sqrtl(d2);
    // If distance between points > diameter, no solution
    if (d > 2*r + EPS || fabsl(d) < EPS) return {};
    P mid = (a + b) / 2;
    ld h2 = r*r - (d/2)*(d/2);
    h2 = max((ld)0.0L, h2);
    P vec = (b - a) / d;
    P perpv = perp(vec); // Perpendicular vector
    P c1 = mid + perpv * sqrtl(h2);
    P c2 = mid - perpv * sqrtl(h2);
    if (c1 == c2) return {c1};
    return {c1, c2};
}
```