Desculpa, Beto - ICPC Library

Contents

1 Data Structures

1.1 Color Update

```
template<class Info>
struct ColorUpdate {
        struct Range {
                int l, r;
          Info value;
                Range(int l = 0, int r = 0, Info value = 0) : l(l), r(r),
                     value(value) {}

                bool operator < (const Range &r) const { return l < r.l; }
        };

        set<Range> ranges;

        void update(int l, int r, Info value) {
                if (l > r) return;

                auto it = ranges.lower_bound(l);
                if (it != ranges.begin()) {
                        it--;
                        if (it->r >= l) {
                                auto cur = *it;
                                ranges.erase(it);
                                ranges.insert(Range(cur.l, l - 1, cur.value
                                    ));
                                ranges.insert(Range(l, cur.r, cur.value));
                        }
                }

                it = ranges.upper_bound(r);
                if (it != ranges.begin()) {
                        it--;
                        if (it->r > r) {
                                auto cur = *it;
                                ranges.erase(it);
                                ranges.insert(Range(cur.l, r, cur.value));
                                ranges.insert(Range(r + 1, cur.r, cur.value
                                    ));
                        }
                }

                ranges.erase(ranges.lower_bound(l), ranges.upper_bound(r));
                ranges.insert(Range(l, r, value));
        }

        Info get(int pos) {
                auto it = ranges.upper_bound(pos);
                if (it == ranges.begin()) return -1;
                it--;
                if (it->r < pos) return -1;
                return it->value;
        }
};
```

1.2 Persistent seg tree

```
const int BUFFER_SIZE = 2e6;

//lembre do vazio
struct node_t {
  node_t *l, *r;

  node_t() {

    l = nullptr;
    r = nullptr;
  }

  node_t(node_t* L, node_t* R) {

    l = L;
    r = R;
  }

  node_t(node_t L, node_t R) {

  }
};

template <typename val_t, typename n_t = node_t>
struct persistent_tree_t {
private:
  vector<n_t*> versions;
  n_t buffer[BUFFER_SIZE];
  int n, buffer_ptr;

  n_t* NewNode() {
    buffer[buffer_ptr] = n_t();
    return &buffer[buffer_ptr++];
  }

  n_t* NewNode(val_t x) {
    buffer[buffer_ptr] = n_t(x);
    return &buffer[buffer_ptr++];
  }

  n_t* NewNode(n_t *L, n_t *R) {
    buffer[buffer_ptr] = n_t(L, R);
```

```
    return &buffer[buffer_ptr++];
  }

  void clear() {
    versions.clear();
    buffer_ptr = 0;
  }

  n_t* build(int l, int r, vector<val_t> &base) {
    if (l == r) {
      return NewNode(base[l]);
    } else {
      int mid = (l + r) >> 1;
      return NewNode(build(l, mid, base), build(mid + 1, r, base));
    }
  }

  n_t* build(int l, int r) {
    if (l == r) {
      return NewNode();
    } else {
      int mid = (l + r) >> 1;
      return NewNode(build(l, mid), build(mid + 1, r));
    }
  }

  //lembre da folha
  n_t* modify(n_t* cur_node, int l, int r, int id, val_t new_val) {
    if (l == r) {
      return ;
    } else {
      int mid = (l + r) >> 1;
      if (mid < id) {
        return NewNode(cur_node->l, modify(cur_node->r, mid + 1, r, id,
            new_val));
      } else {
        return NewNode(modify(cur_node->l, l, mid, id, new_val), cur_node->
            r);
      }
    }
  }

  n_t get(n_t* cur_node, int l, int r, int a, int b) {
    if (l > b || r < a) {
      return n_t();
    }
    if (l >= a && r <= b) {
      return *cur_node;
    }
    int mid = (l + r) >> 1;
    return n_t(get(cur_node->l, l, mid, a, b), get(cur_node->r, mid + 1, r,
        a, b));
  }

public:
  persistent_tree_t(vector<val_t> &base) {
    clear();
    n = (int) base.size() - 1;
    versions.push_back(build(1, n, base));
  }

  persistent_tree_t(int n = 1) {
    clear();
    versions.push_back(build(1, n));
  }

  void re_create(vector<val_t> &base) {
    clear();
    n = (int) base.size() - 1;
    versions.push_back(build(1, n, base));
  }

  void modify(int vers, int id, val_t x) {
    versions.push_back(modify(versions[vers], 1, n, id, x));
  }

  n_t get(int vers, int l, int r) {
    return get(versions[vers], 1, n, l, r);
  }

  int cur_version() {
    return (int) versions.size() - 1;
  }
};
```

1.3 Seg tree

```
template<typename Info>
class SegmentTree {
private:
        int n;
        vector<Info> tree;

        Info merge(Info &l, Info &r) {

        }

public:
        SegmentTree() {}
        SegmentTree(int k) {
                n = k + 1;
                tree.resize(n << 1);
        }
        SegmentTree(vector<Info> v) {
                n = (int) v.size();
                tree.resize(n << 1);
                for(int i = 0; i < n; i++) {
                        tree[i + n] = v[i];
                }
                for(int i = n - 1; i >= 1; i--) {
                        tree[i] = merge(tree[i + i], tree[i + i + 1]);
                }
        }

        void init(int k) {
                n = k + 1;
                tree.resize(n << 1);
        }

        void up(int pos, Info val) {
                pos += n;
                tree[pos] = val;
                while(pos > 1) {
                        pos >>= 1;
                        tree[pos] = merge(tree[pos + pos], tree[pos + pos +
                                1]);
                }
        }

        Info get(int l, int r) {
                Info ans = ; //colocar a base
                for(l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
                        if (l & 1) {
                                ans = merge(ans, tree[l++]);
                        }
                        if (r & 1) {
                                ans = merge(ans, tree[--r]);
                        }
                }
                return ans;
        }
};
```

1.4 Sparse table

```
template<class Info_t>
class SparseTable {
private:
    vector<int> log2;
    vector<vector<Info_t>> table;

    Info_t merge(Info_t &a, Info_t &b) {

    }

public:
    SparseTable(int n, vector<Info_t> v) {
        log2.resize(n + 1);
        log2[1] = 0;
        for (int i = 2; i <= n; i++) {
            log2[i] = log2[i >> 1] + 1;
        }
        table.resize(n + 1);
        for (int i = 0; i < n; i++) {
            table[i].resize(log2[n] + 1);
        }
        for (int i = 0; i < n; i++) {
            table[i][0] = v[i];
        }
        for (int i = 0; i < log2[n]; i++) {
            for (int j = 0; j < n; j++) {
                if (j + (1 << i) >= n) break;
                table[j][i + 1] = merge(table[j][i], table[j + (1 << i)][i
                    ]);
            }
        }
    }

    int get(int l, int r) {
        int k = log2[r - l + 1];
        return merge(table[l][k], table[r - (1 << k) + 1][k]);
    }
};
```

1.5 Sqrt

```
#include <bits/stdc++.h>
using namespace std;

const int block = 500;

struct Node_t {
  vector<char> value;
  int size;

  Node_t() {
    size = 0;
  }
};

int n;
string s;
vector<Node_t> buffer;
vector<int> pos;

void rebuild() {
  pos.clear();
  int i = 0;
  Node_t cur = Node_t();
  while (i < n) {
    if (i % block == 0 && i > 0) {
      pos.push_back(buffer.size());
      buffer.push_back(cur);
      cur = Node_t();
    }
    cur.size++;
    cur.value.push_back(s[i]);
    i++;
  }
  if (cur.size > 0) {
    pos.push_back(buffer.size());
    buffer.push_back(cur);
  }
}

void take() {
  s = "";
  for (int i = 0; i < (int) pos.size(); i++) {
    for (int j = 0; j < buffer[pos[i]].size; j++) {
      s.push_back(buffer[pos[i]].value[j]);
    }
  }
}

int split(int id) {
  int i = 0, on = 0;
  while (i < (int) pos.size() && on < id) {
    on += buffer[pos[i]].size;
    i++;
  }
  if (on == id) {
    return i;
  }
  i--;
  on -= buffer[pos[i]].size;
  Node_t new_l = Node_t(), new_r = Node_t();
  int x = 0;
  while (on < id) {
    new_l.value.push_back(buffer[pos[i]].value[x++]);
    new_l.size++;
    on++;
  }
  while (x < buffer[pos[i]].size) {
    new_r.value.push_back(buffer[pos[i]].value[x++]);
    new_r.size++;
  }
  pos[i] = buffer.size();
  buffer.push_back(new_l);
  pos.insert(pos.begin() + i + 1, buffer.size());
  buffer.push_back(new_r);
  return i + 1;
}

//[l, r], [0...]
void rotate(int l, int r, int k) {
  int id_l = split(l);
  int id_mid = split(l + k);
  int id_r = split(r + 1);
  vector<int> to_put;
  for (int i = id_l; i < id_mid; i++) {
    to_put.push_back(pos[i]);
  }
  for (int i = id_mid; i < id_r; i++) {
    pos[id_l + i - id_mid] = pos[i];
  }
  for (int i = 0; i < id_mid - id_l; i++) {
    pos[id_l + id_r - id_mid + i] = to_put[i];
  }
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
```

```
  cin >> s;
  n = (int) s.size();
  rebuild();
  int m;
  cin >> m;
  vector<int> l(m), r(m), k(m);
  for (int i = 0; i < m; i++) {
    cin >> l[i] >> r[i] >> k[i];
    l[i]--;
    r[i]--;
  }
  for (int i = m - 1; i >= 0; i--) {
    if (i % block == 0) {
      take();
      rebuild();
    }
    rotate(l[i], r[i], k[i]);
  }
  take();
  cout << s << endl;
  return 0;
}
```

1.6 Treap

```
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());

template <typename Info_t>
class Treap {
private:
  struct node_t {
    long long int prior;
    int size;
    Info_t value;
    node_t *l, *r;

    node_t(Info_t a) {
      value = a;
      prior = rng();
      l = nullptr;
      r = nullptr;
      size = 1;
    }
  };

  typedef node_t* pnode_t;

  int sz(pnode_t t) {
    if (t) return t->size;
    return 0;
  }

  void upd_sz(pnode_t t) {
    if (t) {
      t->size = 1 + sz(t->l) + sz(t->r);
    }
  }

  void split(pnode_t cur, Info_t key, pnode_t &l, pnode_t &r) {
    if (cur == nullptr) {
      l = r = nullptr;
    } else if (cur->value < key) {
      l = cur;
      split(cur->r, key, l->r, r);
      upd_sz(l);
    } else {
      r = cur;
      split(cur->l, key, l, r->l);
      upd_sz(r);
    }
  }

  void merge(pnode_t l, pnode_t r, pnode_t &ans) {
    if (l == nullptr) {
      ans = r;
    } else if (r == nullptr) {
      ans = l;
    } else {
      if (l->prior >= r->prior) {
        ans = l;
        merge(l->r, r, ans->r);
      } else {
        ans = r;
        merge(l, r->l, ans->l);
      }
    }
    upd_sz(ans);
  }

public:
  pnode_t root;

  Treap() {
    root = nullptr;
  }

  bool search(Info_t x, pnode_t cur) {
    if (cur == nullptr) {
      return false;
    }
    if (cur->value > x) {
      return search(x, cur->l);
    } else if (cur->value < x) {
      return search(x, cur->r);
    } else {
      return true;
    }
  }

  void insert(Info_t x) {
    //if it cannot contain equal elements
    if (search(x, root)) {
      return;
    }
    pnode_t l, r;
    split(root, x, l, r);
    merge(l, new node_t(x), root);
    merge(root, r, root);
  }

  void erase(Info_t x) {
    pnode_t l, mid, r;
    split(root, x, l, mid);
    split(mid, x + 1, mid, r);
    merge(l, r, root);
    if (mid != nullptr) {
      assert(mid->size == 1 && mid->value == x);
      delete mid;
    }
  }

  int get(Info_t x, pnode_t cur) {
    if (!cur) {
      return 0;
    }
    if (cur->value < x) {
      return 1 + sz(cur->l) + get(x, cur->r);
    } else {
      return get(x, cur->l);
    }
  }
};

//implicit with reversion
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
```

```
template <typename Info_t>
class Treap {
private:
  struct node_t {
    long long int prior;
    int size;
    Info_t value;
    node_t *l, *r;
    bool rev;

    node_t(Info_t a) {
      prior = rng();
      l = nullptr;
      r = nullptr;
      size = 1;
      value = a;
      rev = false;
    }
  };

  typedef node_t* pnode_t;

  int sz(pnode_t t) {
    if (t) return t->size;
    return 0;
  }

  void upd_sz(pnode_t t) {
    if (t != nullptr) {
      t->size = 1 + sz(t->l) + sz(t->r);
    }
  }

  void push(pnode_t t) {
    if (t != nullptr && t->rev) {
      t->rev = false;
      swap(t->l, t->r);
      if (t->l) {
        t->l->rev ^= true;
      }
      if (t->r) {
        t->r->rev ^= true;
      }
    }
  }

  void split(pnode_t cur, int key, int add, pnode_t &l, pnode_t &r) {
    if (cur == nullptr) {
      l = r = nullptr;
      return;
    }
    push(cur);
    int cur_key = add + sz(cur->l);
    if (cur_key < key) {
      l = cur;
      split(cur->r, key, 1 + sz(cur->l) + add, l->r, r);
      upd_sz(r);
      upd_sz(l);
    } else {
      r = cur;
      split(cur->l, key, add, l, r->l);
      upd_sz(r);
      upd_sz(l);
    }
  }

  void merge(pnode_t l, pnode_t r, pnode_t &ans) {
    push(l);
    push(r);
    if (l == nullptr) {
      ans = r;
    } else if (r == nullptr) {
      ans = l;
    } else {
      if (l->prior >= r->prior) {
        ans = l;
        merge(l->r, r, ans->r);
      } else {
        ans = r;
        merge(l, r->l, ans->l);
      }
    }
    upd_sz(ans);
  }

public:
  pnode_t root;

  Treap() {
    root = nullptr;
  }

  bool search(Info_t x, pnode_t cur) {
    if (cur == nullptr) {
      return false;
    }
    if (cur->value > x) {
      return search(x, cur->l);
    } else if (cur->value < x) {
      return search(x, cur->r);
    } else {
      return true;
    }
  }

  void insert(Info_t x, int id = -1) {
    if (id == -1) {
      id = get_less_than(x, root);
    }
    pnode_t l, r;
    split(root, id, 0, l, r);
    merge(l, new node_t(x), root);
    merge(root, r, root);
  }

  void erase(int id) {
    pnode_t l, mid, r;
    split(root, id, 0, l, mid);
    split(mid, 1, 0, mid, r);
    merge(l, r, root);
    if (mid != nullptr) {
      delete mid;
    }
  }

  int get_less_than(Info_t x, pnode_t cur) {
    if (!cur) {
      return 0;
    }
    if (cur->value < x) {
      return 1 + sz(cur->l) + get_less_than(x, cur->r);
    } else {
      return get_less_than(x, cur->l);
    }
  }

  void print(pnode_t cur) {
    if (cur == nullptr) {
      return;
    }
    print(cur->l);
    cout << cur->value << ' ';
    print(cur->r);
  }

  void reverse(int L, int R) {
    pnode_t l, mid, r;
    split(root, L, 0, l, mid);
    split(mid, R - L + 1, 0, mid, r);
    assert(mid->rev == false);
```

```
    mid->rev = true;
    merge(l, mid, root);
    merge(root, r, root);
  }

  Info_t get(int pos) {
    pnode_t l, mid, r;
    split(root, pos, 0, l, mid);
    split(mid, 1, 0, mid, r);
    Info_t ans = mid->value;
    merge(l, mid, root);
    merge(root, r, root);
    return ans;
  }
};
```

2 Graph Algorithms

2.1 arborescence

```
using ll = long long;
using i3 = pair<ll, pair<int, int>>;

struct UnionFind {
  vector<int> par;
  UnionFind(int n) : par(n) {
    for (int i = 0; i < n; ++i) par[i] = i;
  }
  int find(int i) { return (par[i] == i ? i : (par[i] = find(par[i]))); }
  void merge(int i, int j) {
    i = find(i); j = find(j);
    par[j] = i;
  }
};

template <class T>
struct SkewHeap {
  priority_queue<T, vector<T>, greater<T>> heap;
  ll lazy;

  SkewHeap() { lazy = 0; }
  void insert(T v) { heap.push(v); }
  void adjust(ll x) { lazy += x; }
  bool empty() { return heap.empty(); }
  T apply(T ans, ll lazy) {
    ans.first += lazy;
    return ans;
  }
  T pop_min() {
    T ans = heap.top();
    ans = apply(ans, lazy);
    heap.pop();
    return ans;
  }
  void absorb(SkewHeap<T>& o) {
    if (o.heap.size() > heap.size()) {
      swap(o.heap, heap);
      swap(o.lazy, lazy);
    }
    ll diff = o.lazy - lazy;
    while (!o.heap.empty()) {
      T it = o.heap.top();
      o.heap.pop();
      insert(apply(it, diff));
    }
  }
};

struct edge { int to; int v; ll w; int id;};

ll get_arborescence(int n, vector<vector<edge>> &E, vector<edge>& all_edges
    , vector<int>& ant) {
  ant.assign(n, -1);
  UnionFind uf(n);
  vector<SkewHeap<i3>> sk(n);
  for (int i = 1; i < n; ++i)
    for (const edge &e : E[i])
      sk[i].insert({e.w, {e.v, e.id}});
  ll ans = 0LL;
  vector<ll> best(n, -1);
  for (int i = 1, root = 0; i < n; ++i) {
    if (uf.find(i) == root) continue;
    vector<int> st; st.push_back(i);
    while (true) {
      int u = st.back(), v, id; ll w;
      if (sk[u].empty()) {
        return -1; //impossible
      }
      i3 ret = sk[u].pop_min();
      w = ret.first; v = ret.second.first; id = ret.second.second;
      v = uf.find(v);
      if (v == u) continue;
      ant[all_edges[id].to] = id;
      ans += (best[u] = w);
      if (v == root) break;
      if (best[v] == -1) {
        st.push_back(v);
      } else {
        while (true) {
          sk[st.back()].adjust(-best[st.back()]);
          if (st.back() != u) sk[u].absorb(sk[st.back()]);
          if (uf.find(st.back()) == v) break;
          else uf.merge(st.back(), v), v = uf.find(v), st.pop_back();
        }
        swap(sk[u], sk[v]);
        st.pop_back(), st.push_back(v);
      }
    }
    while (!st.empty()) uf.merge(root, st.back()), st.pop_back();
    root = uf.find(root);
  }
  return ans;
}
```

2.2 Centroid Decomposition

```
//Centroid decomposition1

void dfsSize(int v, int pa) {
        sz[v] = 1;
        for(int u : adj[v]) {
                if (u == pa || rem[u]) continue;
                dfsSize(u, v);
                sz[v] += sz[u];
        }
}

int getCentroid(int v, int pa, int tam) {
        for(int u : adj[v]) {
                if (u == pa || rem[u]) continue;
                if (2 * sz[u] > tam) return getCentroid(u, v, tam);
        }
        return v;
}

void decompose(int v, int pa = -1) {
        //cout << v << ' ' << pa << '\n';
        dfsSize(v, pa);
```

```
        int c = getCentroid(v, pa, sz[v]);
        //cout << c << '\n';
        par[c] = pa;
        rem[c] = 1;
        for(int u : adj[c]) {
                if (!rem[u] && u != pa) decompose(u, c);
        }
        adj[c].clear();
}

//Centroid decomposition2

void dfsSize(int v, int par) {
        sz[v] = 1;
        for(int u : adj[v]) {
                if (u == par || removed[u]) continue;
                dfsSize(u, v);
                sz[v] += sz[u];
        }
}

int getCentroid(int v, int par, int tam) {
        for(int u : adj[v]) {
                if (u == par || removed[u]) continue;
                if (2 * sz[u] > tam) return getCentroid(u, v, tam);
        }
        return v;
}

void setDis(int v, int par, int nv, int d) {
        dis[v][nv] = d;
        for(int u : adj[v]) {
                if (u == par || removed[u]) continue;
                setDis(u, v, nv, d + 1);
        }
}

void decompose(int v, int par, int nv) {
        dfsSize(v, par);
        int c = getCentroid(v, par, sz[v]);
        ct[c] = par;
        removed[c] = 1;
        setDis(c, par, nv, 0);
        for(int u : adj[c]) {
                if (!removed[u]) {
                        decompose(u, c, nv + 1);
                }
        }
}
```

3 Dynamic Programming

3.1 CHT

```
typedef long double double_t;
typedef long long int ll;

class HullDynamic {
public:
    const double_t inf = 1e9;

    struct Line {
        ll m, b;
        double_t start;
        bool is_query;

        Line() {}
        Line(ll _m, ll _b, double_t _start, bool _is_query) : m(_m), b(_b),
             start(_start), is_query(_is_query) {}

        ll eval(ll x) {
            return m * x + b;
        }

        double_t intersect(const Line& l) const {
            return (double_t) (l.b - b) / (m - l.m);
        }

        bool operator< (const Line& l) const {
            if (is_query == 0) return m > l.m;
            return (start < l.start);
        }
    };

    typedef set<Line>::iterator iterator_t;

    bool has_prev(iterator_t it) {
        return (it != hull.begin());
    }

    bool has_next(iterator_t it) {
        return (++it != hull.end());
    }

    bool irrelevant(iterator_t it) {
        if (!has_prev(it) || !has_next(it)) return 0;
        iterator_t prev = it, next = it;
        prev--;
        next++;
        return next->intersect(*prev) <= it->intersect(*prev);
    }

    void update_left(iterator_t it) {
        if (it == hull.begin()) return;
        iterator_t pos = it;
        --it;
        vector<Line> rem;
        while(has_prev(it)) {
            iterator_t prev = it;
            --prev;
            if (prev->intersect(*pos) <= prev->intersect(*it)) {
                rem.push_back(*it);
            } else {
                break;
            }
            --it;
        }
        double_t start = pos->intersect(*it);
        Line f = *pos;
        for (Line r : rem) hull.erase(r);
        hull.erase(f);
        f.start = start;
        hull.insert(f);
    }

    void update_right(iterator_t it) {
        if (!has_next(it)) return;
        iterator_t pos = it;
        ++it;
        vector<Line> rem;
        while(has_next(it)) {
            iterator_t next = it;
            ++next;
            if (next->intersect(*pos) <= pos->intersect(*it)) {
                rem.push_back(*it);
            } else {
                break;
            }
            ++it;
        }
```

```
        double_t start = pos->intersect(*it);
        Line f = *it;
        for (Line r : rem) hull.erase(r);
        hull.erase(f);
        f.start = start;
        hull.insert(f);
    }

    void insert_line(ll m, ll b) {
        Line f(m, b, -inf, 0);
        iterator_t it = hull.lower_bound(f);
        if (it != hull.end() && it->m == f.m) {
            if (it->b <= f.b) {
                return;
            } else if (it->b > f.b) {
                hull.erase(it);
            }
        }
        hull.insert(f);
        it = hull.lower_bound(f);
        if (irrelevant(it)) {
            hull.erase(it);
            return;
        }
        update_left(it);
        it = hull.lower_bound(f);
        update_right(it);
    }

    ll get(ll x) {
        Line f(0, 0, x, 1);
        iterator_t it = hull.upper_bound(f);
        assert(it != hull.begin());
        --it;
        return it->m * x + it->b;
    }

private:
    set<Line> hull;
};


////////////////////////////////
//Linear

class HullDynamic {
public:
    const double_t inf = 1e18;

    struct Line {
        ll m, b;
        double_t start;
        bool is_query;

        Line() {}

        Line(ll _m, ll _b, double_t _start, bool _is_query) : m(_m), b(_b),
             start(_start), is_query(_is_query) {}

        ll eval(ll x) {
            return m * x + b;
        }

        double_t intersect(const Line& l) const {
            return (double_t) (l.b - b) / (m - l.m);
        }

        bool operator< (const Line& l) const {
            if (is_query == 0) return m > l.m;
            return (start < l.start);
        }
    };

    typedef vector<Line>::iterator iterator_t;

    void insert(ll m, ll b) {
        Line f(m, b, -inf, 0);
        while((int) hull.size() > 1 && f.intersect(hull[(int) hull.size() -
              2]) <= hull.back().start) {
            hull.pop_back();
        }
        double_t st = -inf;
        if (!hull.empty()) {
            st = f.intersect(hull.back());
        }
        f.start = st;
        hull.push_back(f);
        /*for (auto it : hull) {
            cout << it.m << ' ' << it.b << ' ' << it.start << '\n';
        }
        cout << '\n';*/
    }

    ll get(ll x) {
        pt = min(pt, (int) hull.size() - 1);
        while(pt < (int) hull.size() - 1) {
            if (hull[pt + 1].start <= x) {
                pt++;
            } else {
                break;
            }
        }
        //cout << hull[pt].m << ' ' << hull[pt].b << '\n';
        return hull[pt].m * x + hull[pt].b;
    }

private:
    vector<Line> hull;
};
```

3.2 Slope Trick

```
#include <limits>
#include <queue>
#include <utility>

// Thanks WeakestTopology

template <typename T>
struct SlopeTrick {
  static constexpr T inf = std::numeric_limits<T>::max() / 2;

  T cost = 0, left_offset = 0, right_offset = 0;
  using Pair = std::pair<T, T>;
  std::priority_queue<Pair> left;
  std::priority_queue<Pair, std::vector<Pair>, std::greater<Pair>> right;

  SlopeTrick() {
    left.emplace(-inf, 0);
    right.emplace(+inf, 0);
  }

  void rebalance() {
    while (true) {
      auto [xl, sl] = left.top();
      xl -= left_offset;
      auto [xr, sr] = right.top();
      xr += right_offset;
      if (xl <= xr) break;
      left.pop();
      right.pop();
      T take = std::min(sl, sr);
      cost += take * (xl - xr);
      sl -= take;
```

```
      sr -= take;
      left.emplace(xr + left_offset, take);
      right.emplace(xl - right_offset, take);
      if (sl) {
        left.emplace(xl + left_offset, sl);
      }
      if (sr) {
        right.emplace(xr - right_offset, sr);
      }
    }
  }

  // Adds f(x) = s * max(a - x, 0).
  void add_left(T a, T s) {
    left.emplace(a + left_offset, s);
    rebalance();
  }

  // Adds f(x) = s * max(x - a, 0).
  void add_right(T a, T s) {
    right.emplace(a - right_offset, s);
    rebalance();
  }

  // Adds f(x) = s * abs(x - a).
  void add_abs(T a, T s) {
    add_left(a, s);
    add_right(a, s);
  }

  void relax_left(T offset) {
    assert(offset >= 0);
    left_offset += offset;
  }

  void relax_right(T offset) {
    assert(offset >= 0);
    right_offset += offset;
  }

  void relax(T offset) {
    assert(offset >= 0);
    left_offset += offset;
    right_offset += offset;
  }
};
```

4 Math

4.1 Dirichlet Convolution

```
#include <bits/stdc++.h>
using namespace std;

/*
  fib(m + n) = fib(m) * fib(n + 1) + fib(m - 1) * fib(n)
  gcd(fib(n), fib(m)) = fib(gcd(n, m))
*/

/*
Dirichlet Convolution

(f * g)(n) = sum{d | n}{f(d) * g(n / d)}
mi * 1 = e => (g = f * 1 <=> f = g * mi)
phi * 1 = Id
sf(n) = (s(f*g)(n) - sum{2 <= d <= n}{sf(floor(n / d)) * g(d)}) / g(1)
*/
struct ModularArithmeticFunctionPrefixSum {
  unordered_map<long long, long long> dp;
  vector<long long> precalculatedPrefixSumF;
  long long mod;

  ModularArithmeticFunctionPrefixSum(vector<long long>
      precalculatedPrefixSumF, long long mod) : precalculatedPrefixSumF(
      precalculatedPrefixSumF), mod(mod) {}

  long long fexp(long long b, long long e = -1) {
    if (e < 0) e += mod - 1;
    long long ans = 1;
    while (e > 0) {
      if (e & 1) ans = ans * b % mod;
      b = b * b % mod;
      e >>= 1;
    }
    return ans;
  }

  long long prefixG(long long x) {
    // implement here s_g(x)
    // example:
    // g = 1
    if (x <= 0) {
      return 0;
    }
    return x % mod;
  }

  const long long inv2 = fexp(2);

  long long prefixFG(long long x) {
    // implement here s_(f * g)(x)
    // example:
    // f = phi
    // g = 1
    // f * g = Id
    if (x <= 0) {
      return 0;
    }
    x %= mod;
    return x * (x + 1) % mod * inv2 % mod;
  }

  long long prefixF(long long x) {
    if (x < precalculatedPrefixSumF.size()) {
      return precalculatedPrefixSumF[x];
    }
    if (dp.count(x)) {
      return dp[x];
    }
    long long ans = prefixFG(x);
    for (long long lowerBound = 2, upperBound; lowerBound <= x; lowerBound
        = upperBound + 1) {
      upperBound = x / (x / lowerBound);
      long long intervalG = (prefixG(upperBound) - prefixG(lowerBound - 1)
          + mod) % mod;
      ans = (ans - prefixF(x / lowerBound) % mod * intervalG % mod + mod) %
          mod;
    }
    long long g1 = (prefixG(1) - prefixG(0) + mod) % mod;
    ans = ans * fexp(g1) % mod;
    assert(g1 * fexp(g1) % mod == 1);
    dp[x] = ans;
    return ans;
  }
};

const long long mod = 998244353L;
long long acc[2][2], base[2][2], temp[2][2];

long long fib(long long x) {
  if (x == 0) return 0;

  base[0][0] = 1;
  base[0][1] = 1;
  base[1][0] = 1;
```

```
  base[1][1] = 0;
  acc[0][0] = 1;
  acc[0][1] = 0;
  acc[1][0] = 0;
  acc[1][1] = 1;

  x--;
  while (x > 0) {
    if (x & 1) {
      for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
          temp[i][j] = 0;
          for (int k = 0; k < 2; k++) {
            temp[i][j] = (temp[i][j] + base[i][k] * acc[k][j] % mod) % mod;
          }
        }
      }
      for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
          acc[i][j] = temp[i][j];
        }
      }
    }

    x >>= 1;

    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 2; j++) {
        temp[i][j] = 0;
        for (int k = 0; k < 2; k++) {
          temp[i][j] = (temp[i][j] + base[i][k] * base[k][j] % mod) % mod;
        }
      }
    }
    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 2; j++) {
        base[i][j] = temp[i][j];
      }
    }
  }

  return acc[0][0];
}

int main() {
  ios::sync_with_stdio(false); cin.tie(nullptr);

  const int mx = 5000000;
  vector<int> primes, phi(mx);
  vector<bool> isComposite(mx, false);

  phi[1] = 1;

  for (int i = 2; i < mx; i++) {
    if (!isComposite[i]) {
      primes.push_back(i);
      phi[i] = i - 1;
    }
    //if (i < 30) cout << i << ' ' << phi[i] << endl;
    for (int p: primes) {
      if (i * p >= mx) break;
      isComposite[i * p] = true;
      if (i % p == 0) {
        phi[i * p] = p * phi[i];
        break;
      } else {
        phi[i * p] = phi[i] * phi[p];
      }
    }
  }

  vector<long long> sumPhi(mx);
  sumPhi[0] = phi[0];
  for (int i = 1; i < mx; i++) {
    sumPhi[i] = (1ll * phi[i] + sumPhi[i - 1]) % mod;
  }

  ModularArithmeticFunctionPrefixSum phiCalculator(sumPhi, mod);

  //cout << phiCalculator.prefixF(1ll * 10000000000) << endl;

  long long n;
  cin >> n;

  long long ans = 0;
  for (long long lowerBound = 1, upperBound; lowerBound <= n; lowerBound =
      upperBound + 1) {
    upperBound = n / (n / lowerBound);
    long long fibSum = (fib(upperBound + 2) - 1 - fib(lowerBound + 1) + 1 +
        mod) % mod;
    long long lim = n / lowerBound;
    assert(n / lowerBound == n / upperBound);

    ans = (ans + (2 * phiCalculator.prefixF(lim) - 1 + mod) % mod * fibSum
        % mod) % mod;
  }

  cout << ans << endl;

  return 0;
}
```

4.2 Miller and Rho

```
typedef long long int ll;

bool overflow(ll a, ll b) {
        return b && (a >= (1ll << 62) / b);
}

ll add(ll a, ll b, ll md) {
        return (a + b) % md;
}

ll mul(ll a, ll b, ll md) {
        if (!overflow(a, b)) return (a * b) % md;
        ll ans = 0;
        while(b) {
                if (b & 1) ans = add(ans, a, md);
                a = add(a, a, md);
                b >>= 1;
        }
        return ans;
}

ll fexp(ll a, ll e, ll md) {
        ll ans = 1;
        while(e) {
                if (e & 1) ans = mul(ans, a, md);
                a = mul(a, a, md);
                e >>= 1;
        }
        return ans;
}

ll my_rand() {
        ll ans = rand();
        ans = (ans << 31) | rand();
        return ans;
}

ll gcd(ll a, ll b) {
        while(b) {
                ll t = a % b;
                a = b;
                b = t;
```

```
        }
        return a;
}

bool miller(ll p, int iteracao) {
        if(p < 2) return 0;
        if(p % 2 == 0) return (p == 2);
        ll s = p - 1;
        while(s % 2 == 0) s >>= 1;
        for(int i = 0; i < iteracao; i++) {
                ll a = rand() % (p - 1) + 1, temp = s;
                ll mod = fexp(a, temp, p);
                while(temp != p - 1 && mod != 1 && mod != p - 1) {
                        mod = mul(mod, mod, p);
                        temp <<= 1;
                }
                if(mod != p - 1 && temp % 2 == 0) return 0;
        }
        return 1;
}

ll rho(ll n) {
        if (n == 1 || miller(n, 10)) return n;
        if (n % 2 == 0) return 2;
        while(1) {
                ll x = my_rand() % (n - 2) + 2, y = x;
                ll c = 0, cur = 1;
                while(c == 0) {
                        c = my_rand() % (n - 2) + 1;
                }
                while(cur == 1) {
                        x = add(mul(x, x, n), c, n);
                        y = add(mul(y, y, n), c, n);
                        y = add(mul(y, y, n), c, n);
                        cur = gcd((x >= y ? x - y : y - x), n);
                }
                if (cur != n) return cur;
        }
}
```

5 String Algorithms

5.1 Hash

```
using ull = uint64_t;

const ull MOD = (1LL << 61) - 1;
const int base = 31;
const int MAX_POT = 1000100;

ull p[MAX_POT];

ull modMul(uint64_t a, uint64_t b) {
  ull l1 = (uint32_t)a, h1 = a >> 32, l2 = (uint32_t)b, h2 = b >> 32;
  ull l = l1 * l2, m = l1 * h2 + l2 * h1, h = h1 * h2;
  ull ret = (l & MOD) + (l >> 61) + (h << 3) + (m >> 29) + ((m << 35) >> 3)
       + 1;
  ret = (ret & MOD) + (ret >> 61);
  ret = (ret & MOD) + (ret >> 61);
  return ret - 1;
}

void buildHashPot() {
  p[0] = 1;
  for (int i = 1; i < MAX_POT; i++) p[i] = modMul(base, p[i - 1]);
}

struct Hash {
  ull val;
  int size;
  Hash(ull val = 0, int size = 0): val(val), size(size) {}
};

// maior pot vai ficar no l
Hash combine(Hash l, Hash r) {
  ull val = modMul(p[r.size], l.val);
  return Hash(val + r.val, l.size + r.size);
}

struct StringHash {
  // 0 based
  // pref[i] = hash da string 0..i, maior pot ta no 0
  // suff[i] = hash da string i..n-1, maior pot ta no n-1
  int n;
  vector<Hash> pref, suff;

  ull getInt(char c) {
    return c - 'a' + 1;
  }

  StringHash(string &s) {
    // remember to call it in the main
    // buildHashPot();
    n = s.size();
    pref.resize(n);
    suff.resize(n);
    pref[0] = Hash(getInt(s[0]), 1);
    for (int i = 1; i < n; i++) {
      Hash cur = Hash(getInt(s[i]), 1);
      pref[i] = combine(pref[i - 1], cur);
    }
    suff[n - 1] = Hash(getInt(s[n - 1]), 1);
    for (int i = n - 2; i >= 0; i--) {
      Hash cur = Hash(getInt(s[i]), 1);
      suff[i] = combine(suff[i + 1], cur);
    }
  }

  // [l, r]
  // maior pot vai estar no l
  Hash getHash(int l, int r) {
    ull res = pref[r].val;
    if(l > 0) res = (res + MOD - modMul(p[r - l + 1], pref[l - 1].val)) %
        MOD;
    return Hash(res, r - l + 1);
  }

  // [l, r]
  // maior pot vai estar no r
  Hash getRevHash(int l, int r) {
    ull res = suff[l].val;
    if (r < n - 1) res = (res + MOD - modMul(p[r - l + 1], suff[r + 1].val)
        ) % MOD;
    return Hash(res, r - l + 1);
  }
};
```

5.2 Kmp Automaton

```
const int limit =

vector<vector<int>> build_automaton(string s) {
    s += '#'; //tem que ser diferente de todos os caracteres
    int n = (int) s.size();
    vector<vector<int>> ans(n, vector<int>(limit));
    vector<int> fail(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < limit; j++) {
            if (i == 0) {
                if (s[i] == j + 'a') {
```

```
                    ans[i][j] = i + 1;
                } else {
                    ans[i][j] = 0;
                }
            } else {
                if (s[i] == j + 'a') {
                    ans[i][j] = i + 1;
                } else {
                    ans[i][j] = ans[fail[i - 1]][j];
                }
            }
        }
        if (i == 0) {
            continue;
        }
        int j = fail[i - 1];
        while (j > 0 && s[i] != s[j]) {
            j = fail[j - 1];
        }
        fail[i] = j + (s[i] == s[j]);
    }
    return ans;
}
```

6 Miscellaneous

6.1 Operations

```
inline int add(int a, int b) {
    a += b;
    if (a >= mod) {
        a -= mod;
    }
    return a;
}

inline int sub(int a, int b) {
    a -= b;
    if (a < 0) {
        a += mod;
    }
    return a;
}

inline int mul(int a, int b) {
    int ans = 0;
    asm(
        "mull %%ebx;"
        "divl %3;"
        : "=d" (ans)
        : "a" (a), "b" (b), "c" (mod)
    );
    return ans;
}
```

6.2 Count Inversions

```
typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;

/*
Without collisions. If there are collisions, it might be possible to use
    less_equal as a comparator (the delete does not work as expected)
*/
int solve(vector<pair<int, int>> all) {
  sort(all.begin(), all.end());
  reverse(all.begin(), all.end());
  int ans = 0;
  ordered_set S;
  for (int i = 0; i < n; i++) {
    ans += S.order_of_key(all[i].second);
    S.insert(all[i].second);
  }
  return ans;
}
```

7 Teoremas e formulas uteis

7.1 Teoremas

```
Quadrangular inequality
- SE cost(l, r) + cost(l + 1, r + 1) <= cost(l, r + 1) + cost(l + 1, r)
- E f(l, r) = dp[l - 1] + cost(l, r)
- Entao f(l, r) tbm obedece a quadrangular inequality
- Isso implica que se existe um k tal que f(l + 1, k) <= f(l, k), entao f(l
    + 1, k + 1) <= f(l, k)
  Ou seja, a partir do momento que o (l + 1) fica melhor do que o l, ele
      sempre eh melhor.
- Com isso da pra otimizar dp[x] para O(n * log), fazendo uma busca binaria
    pra descobrir quando (l + 1) fica melhor do que l

Se e >= log2(m)
n^e mod m = n^(phi(m) + e mod phi(m)) mod m
Se e < log2(m), so fazer o bruto

phi(phi(m)) <= m/2
```