TEAM DEAD TEMPLATES

mhtkrag

October 28, 2023

Contents

1 data-structures

1.1 dsu

```
struct DSU {
  vector<int> parent, siz;
  void init(int n) {
    parent.resize(n);
    siz.resize(n);
    for (int i = 0; i < n; i++) {
      parent[i] = i;
      siz[i] = 1;
    }
  }
  int find(int x) {
    if (x == parent[x]) return x;
    return parent[x] = find(parent[x]);
  }
  void merge(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return;
    if (siz[x] < siz[y]) swap(x, y);
    parent[y] = x;
    siz[x] += siz[y];
  }
  int size(int x) { return siz[find(x)]; }
  bool same(int x, int y) { return find(x) == find(y); }
}
```

1.2 lazy-segment-tree

```
template <typename node_type, typename tag_type>
struct lazy_segtree {
  vector<node_type> tree;
  vector<tag_type> lazy;
  int n;
  template <typename Iter>
  void init(Iter first, Iter last, int nn = -1) {
    n = nn;
    if (n == -1) n = distance(first, last);
    tree.resize(4 * n);
    lazy.resize(4 * n);
    build_tree(0, 0, n - 1, first);
  }
  node_type query(int ql, int qr) { return query(0, 0,
      n - 1, ql, qr); }
  void update(int ql, int qr, tag_type const &val) {
    update(0, 0, n - 1, ql, qr, val);
  }

 private:
  template <typename Iter>
  void build_tree(int id, int tl, int tr, Iter first) {
    if (tl == tr) {
      tree[id].init(tl, tr, *(first + tl));
      lazy[id].init(tl, tr);
      return;
    }
    int tm = (tl + tr) / 2;
    build_tree(2 * id + 1, tl, tm, first);
    build_tree(2 * id + 2, tm + 1, tr, first);
    tree[id] = node_type::merge(tree[2 * id + 1],
        tree[2 * id + 2]);
    lazy[id].init(tl, tr);
  }
  void push(int id, int tl, int tr) {
    if (tl != tr) {
      int tm = (tl + tr) / 2;
      tree[2 * id + 1].apply(tl, tm, lazy[id]);
      lazy[2 * id + 1].merge(lazy[id]);
      tree[2 * id + 2].apply(tm + 1, tr, lazy[id]);
      lazy[2 * id + 2].merge(lazy[id]);
    }
    lazy[id].reset();
  }
  node_type query(int id, int tl, int tr, int ql, int
      qr) {
    if (tl > qr || ql > tr) return node_type::phi();
    if (ql <= tl && tr <= qr) return tree[id];
    push(id, tl, tr);
    int tm = (tl + tr) / 2;
    return node_type::merge(query(2 * id + 1, tl, tm,
        ql, qr),
                            query(2 * id + 2, tm + 1, tr,
                                ql, qr));
  }
  void update(int id, int tl, int tr, int ql, int qr,
      tag_type const &val) {
    if (tl > qr || ql > tr) return;
    if (ql <= tl && tr <= qr) {
      tree[id].apply(tl, tr, val);
      lazy[id].merge(val);
      return;
    }
    push(id, tl, tr);
    int tm = (tl + tr) / 2;
    update(2 * id + 1, tl, tm, ql, qr, val);
    update(2 * id + 2, tm + 1, tr, ql, qr, val);
    tree[id] = node_type::merge(tree[2 * id + 1],
        tree[2 * id + 2]);
  }
};
struct tag {
  ll inc;
  void reset() { inc = 0; }
  void merge(tag const &other) { inc += other.inc; }
  void init(int tl, int tr) {}
};
struct node {
  static node phi() { return {0LL}; }
  static node merge(node const &a, node const &b) {
      return {a.data + b.data}; }
  ll data;
  void apply(ll tl, ll tr, tag const &t) { data += (tr
      - tl + 1) * t.inc; }
  template <typename T>
  void init(ll tl, ll tr, T &ddata) {
    this->data = ddata;
  }
};
```

1.3 segment-tree

```
template <typename T, typename CombineT>
struct SegmentTree {
  vector<T> tree;
  CombineT combine;
  T defaultValue;
  ll n;
```

```
  void init(ll n, T val) {
    tree.resize(4 * n);
    this->n = n;
    defaultValue = val;
    build_tree(1, 0, n - 1);
  }

  template <typename Itr>
  void init(Itr begin, Itr end) {
    n = distance(begin, end);
    tree.resize(4 * n);

    build_tree(1, 0, n - 1, begin);
  }

  void build_tree(ll id, ll tl, ll tr) {
    if (tl == tr) {
      tree[id] = defaultValue;
      return;
    }
    ll tm = (tl + tr) / 2;
    build_tree(id * 2, tl, tm);
    build_tree(id * 2 + 1, tm + 1, tr);
    tree[id] = combine(tree[id * 2], tree[id * 2 + 1]);
  }

  template <typename Itr>
  void build_tree(ll id, ll tl, ll tr, Itr begin) {
    if (tl == tr) {
      tree[id] = *(begin + tl);
      return;
    }
    ll tm = (tl + tr) / 2;
    build_tree(id * 2, tl, tm, begin);
    build_tree(id * 2 + 1, tm + 1, tr, begin);
    tree[id] = combine(tree[id * 2], tree[id * 2 + 1]);
  }

  T query(ll id, ll tl, ll tr, ll ql, ll qr) {
    if (ql > tr || tl > qr) return defaultValue;
    if (ql <= tl && tr <= qr) return tree[id];
    ll tm = (tl + tr) / 2;
    return combine(query(id * 2, tl, tm, ql, qr),
                   query(id * 2 + 1, tm + 1, tr, ql, qr));
  }

  T query(ll l, ll r) { return query(1, 0, n - 1, l,
      r); }

  void update(ll id, ll tl, ll tr, ll p, T x) {
    if (tl == tr) {
      tree[id] = x;
      return;
    }
    ll tm = (tl + tr) / 2;
    if (p <= tm)
      update(id * 2, tl, tm, p, x);
    else
      update(id * 2 + 1, tm + 1, tr, p, x);
    tree[id] = combine(tree[id * 2], tree[id * 2 + 1]);
  }

  void update(ll p, T x) { update(1, 0, n - 1, p, x); }
};
```

1.4 sparse-table

```
struct min_op {
  ll operator()(ll a, ll b) { return min(a, b); }
};
struct max_op {
  ll operator()(ll a, ll b) { return max(a, b); }
};
struct gcd_op {
  ll operator()(ll a, ll b) { return __gcd(a, b); }
};
template <typename OperationT>
struct sparse_table {
  vector<vector<ll>> m;
  OperationT op;
  template <typename Itr>
  void init(Itr begin, Itr end) {
    ll sz = end - begin;
    ll lg = 63 - __builtin_clzll(sz);
    m.assign(sz, vector<ll>(lg + 1));
    for (ll j = 0; j <= lg; ++j) {
      ll len = (1 << j);
      for (ll i = 0; i + len - 1 < sz; ++i) {
        if (len == 1) {
          m[i][j] = *(begin + i);
        } else {
          m[i][j] = op(m[i][j - 1], m[i + (1 << (j -
              1))][j - 1]);
        }
      }
    }
  }
  ll query(ll L, ll R) {
    ll j = 63 - __builtin_clzll((R - L + 1));
    return op(m[L][j], m[R + 1 - (1 << j)][j]);
  }
};
```

2 misc

2.1 bitwise-tricks

```
// iterating over subsets
for (int x = mask;; x = (x - 1) & mask) {
  // Code here...
  if (x == 0) break;
}
```

2.2 cppt

```
#include <bits/stdc++.h>
using namespace std;
#define all(x) begin(x), end(x)
#define OUT(T) cout << "Case #" << T << ": "
#ifndef _DEBUG
#define endl '\n'
#endif
#ifdef _DEBUG
void dbg_out() { cerr << endl; }
template <typename Head, typename... Tail>
void dbg_out(Head H, Tail... T) {
  cerr << ' ' << H;
```

```
  dbg_out(T...);
}
#define dbg(...) cerr << "(" << #__VA_ARGS__ << "):",
    dbg_out(__VA_ARGS__)
#else
#define dbg(...)
#endif
#define ckmin(x, y) x = min((x), (y))
#define ckmax(x, y) x = max((x), (y))

// clang-format off
template <typename T> ostream &operator<<(ostream &out,
    const vector<T> &v) { for (const auto &x : v) out
    << x << ' '; return out; }
template <typename T> istream &operator>>(istream &in,
    vector<T> &v) { for (auto &x : v) in >> x; return
    in; }
// clang-format on

using ll = long long;
using lld = long double;
using pll = pair<ll, ll>;
using pii = pair<int, int>;

void solve(ll _t) {}

int main() {
  ios_base::sync_with_stdio(false), cin.tie(NULL);

  ll T = 1;
  cin >> T;
  for (ll t = 1; t <= T; ++t) solve(t);
}
```

2.3 ordered-set

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T, typename ComparatorFn = less<T>>
using ordered_set = tree<T, null_type, ComparatorFn,
    rb_tree_tag,
                         tree_order_statistics_node_update>;
```

3 number-theory

3.1 modular-int

```
const ll MOD = 1e9 + 7;
ll binexp(ll a, ll b, ll p = MOD) {
  if (b < 0) return 0;
  ll res = 1;
  while (b > 0) {
    if (b & 1) b--, res = (res * a) % p;
    a = (a * a) % p;
    b >>= 1;
  }
  return res;
}

inline ll modinv(ll x, ll p = MOD) { return binexp(x, p
    - 2, p); }
template <ll mod>
struct mi_ {
  ll value;
  mi_() = default;
  mi_(ll x) : value(x % mod) {}
  mi_(const mi_ &m) : value(m.value % mod) {}
  mi_ &operator=(const mi_ &m) {
    value = m.value;
    return *this;
  }
  ll inverse_value() const { return modinv(value, mod);
      }
  mi_ &operator+=(const mi_ &m) {
    value = (value + m.value) % mod;
    return *this;
  }
  mi_ &operator-=(const mi_ &m) {
    value = (mod + value - m.value) % mod;
    return *this;
  }
  mi_ &operator*=(const mi_ &m) {
    value = (value * m.value) % mod;
    return *this;
  }
  mi_ &operator/=(const mi_ &m) {
    value = (value * m.inverse_value()) % mod;
    return *this;
  }
  mi_ &operator++() {
    value++;
    value %= mod;
    return *this;
  }
  mi_ &operator--() {
    value--;
    value %= mod;
    return *this;
  }
  mi_ operator*(const mi_ &b) { return mi_(value *
      b.value); }
  mi_ operator*(ll b) { return mi_(value * b); }
  mi_ operator-(const mi_ &b) { return mi_(mod + value
      - b.value); }
  mi_ operator-(ll b) { return mi_(mod + value - b); }
  mi_ operator+(const mi_ &b) { return mi_(value +
      b.value); }
  mi_ operator+(ll b) { return mi_(value + b); }
  mi_ operator/(const mi_ &b) { return mi_(value *
      modinv(b.value, mod)); }
  mi_ operator/(ll b) { return mi_(value * modinv(b,
      mod)); }
};
template <ll mod>
ostream &operator<<(ostream &out, const mi_<mod> &m) {
  out << m.value % mod;
  return out;
}
template <ll mod>
istream &operator>>(istream &in, mi_<mod> &m) {
  ll x;
  in >> x;
  m.value = (x % mod);
  return in;
}
using mi = mi_<MOD>;
vector<mi> factorial;
void init_factorial() {
  factorial.resize(1000005);
```

```
  factorial[0] = factorial[1] = 1;
  for (ll i = 2; i < 1000005; ++i) {
    factorial[i] = (factorial[i - 1] * i);
  }
}
inline mi choose(const mi &a, const mi &b) {
  if (a.value < b.value) return 0;
  return factorial[a.value] /
         (factorial[b.value] * factorial[(a.value -
             b.value)]);
}
```

3.2 sieve

```
struct sieve_t {
  sieve_t(int n, bool gen_primes = false, bool
      gen_sieve = false) {
    is_prime.assign(n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i * i <= n; ++i) {
      for (int j = i * i; j <= n; j += i)
        is_prime[j] = false;
    }
    if (gen_primes) {
      for (int i = 2; i <= n; ++i) {
        if (is_prime[i])
          primes.push_back(i);
      }
    }
    if (gen_sieve) {
      sieve.assign(n + 1, -1);
      for (int i = 2; i <= n; ++i) {
        if (is_prime[i]) {
          sieve[i] = i;
          if ((ll)i * i <= n) {
            for (int j = i * i; j <= n; j += i) {
              if (sieve[j] == -1)
                sieve[j] = i;
            }
          }
        }
      }
    }
  }
  // requires gen_fact; works only upto sz;
  vector<int> fast_factorize(int k) {
    vector<int> res;
    while (k > 1) {
      ll p = sieve[k];
      res.push_back(p);
      k /= p;
    }
    return res;
  }
  // requies gen_primes; works upto sz*sz;
  vector<int> factorize(int k) {
    vector<int> res;
    for (int p : primes) {
      if (p * p > k)
        break;
      while (k % p == 0) {
        k /= p;
        res.push_back(p);
      }
    }
    if (k > 1)
      res.push_back(k);
    return res;
  }
  vector<bool> is_prime;
  vector<int> primes;
  vector<int> sieve;
};
```

4 strings

4.1 kmp

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

// pi[i] = longest proper prefix of s[0..i] which is
    alos a suffix;
// online algorithm;
vector<int> prefix_function(string const& s) {
  int n = s.length();
  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;
  }
}

// Applications:

// finding occurences: Concat 's # t' and check in
    where pi[i] = |S|
// counting prefixes;
vector<int> prefix_occurences(vector<int> const& pi,
    int n) {
  vector<int> ans(n + 1);
  for (int i = 0; i < n; i++) ans[pi[i]]++;
  for (int i = n - 1; i > 0; i--) ans[pi[i - 1]] +=
      ans[i];
  for (int i = 0; i <= n; i++) ans[i]++;
  return ans;
}

// compression, if k = n - pi[n-1], divides n then 'k'
    is smallest
// unit to compress the string 's';

void compute_automaton(string s, vector<vector<int>>&
    aut) {
  s += '#';
  int n = s.size();
  vector<int> pi = prefix_function(s);
  aut.assign(n, vector<int>(26));
  for (int i = 0; i < n; i++) {
    for (int c = 0; c < 26; c++) {
      if (i > 0 && 'a' + c != s[i])
        aut[i][c] = aut[pi[i - 1]][c];
      else
        aut[i][c] = i + ('a' + c == s[i]);
    }
  }
```

```
}
```

5 tree

5.1 binary-lifting

```
struct binary_lift {
  vector<vector<ll>> children;
  vector<ll> depth;
  const ll LOG = 18;
  void init(vector<vector<ll>> &adj) {
    ll n = adj.size();
    depth.resize(n);
    children.assign(n, vector<ll>(LOG + 1));
    function<void(ll, ll, ll)> dfs = [&](ll u, ll p, ll
        d) {
      depth[u] = d;
      children[u][0] = p;
      for (ll i = 1; i <= LOG; ++i) {
        children[u][i] = children[children[u][i - 1]][i
            - 1];
      }
      for (ll v : adj[u]) {
        if (v != p)
          dfs(v, u, d + 1);
      }
    };
    dfs(0, 0, 0);
  }
  ll lift_node(ll n, ll d) {
    for (ll i = LOG; i >= 0; --i) {
      if (d & (1 << i))
        n = children[n][i];
    }
    return n;
  }
  ll lca(ll u, ll v) {
    if (depth[u] < depth[v])
      swap(u, v);
    u = lift_node(u, depth[u] - depth[v]);
    if (u == v)
      return u;
    for (ll i = LOG; i >= 0; --i) {
      if (children[u][i] != children[v][i]) {
        u = children[u][i];
        v = children[v][i];
      }
    }
    return children[u][0];
  }
  ll dist(ll u, ll v) { return depth[u] + depth[v] - 2
      * depth[lca(u, v)]; }
};
```

5.2 euler-tour

```
struct euler_tour {
  vector<ll> in, out;
  ll timer = 0;
  void init(vector<vector<ll>> &adj) {
    ll n = adj.size();
    in.resize(n);
    out.resize(n);
    function<void(ll, ll)> dfs = [&](ll u, ll p) {
      in[u] = timer++;
      for (ll v : adj[u]) {
        if (v != p)
          dfs(v, u);
      }
      out[u] = timer++;
    };
    dfs(0, 0);
  }
  bool is_ancestor(ll u, ll v) { return in[u] <= in[v]
      && out[u] >= out[v]; }
};
```

5.3 tree-lifing

```
/*
 * Does all the binary lifting tasks in the same
 * time complexity but uses only O(n) memory;
 */
struct tree_lifting {
  vector<int> dep, jmp, fa;
  int n;
  void add_leaf(int cur, int par) {
    fa[cur] = par;
    dep[cur] = 1 + dep[par];
    if (dep[par] - dep[jmp[par]] == dep[jmp[par]] -
        dep[jmp[jmp[par]]]) {
      jmp[cur] = jmp[jmp[par]];
    } else {
      jmp[cur] = par;
    }
  }
  void dfs(int cur, int par, vector<vector<int>> &adj) {
    add_leaf(cur, par);
    for (int it : adj[cur]) {
      if (it == par)
        continue;
      dfs(it, cur, adj);
    }
  }
  void init(int nn) {
    dep.resize(nn), jmp.resize(nn), fa.resize(nn);
    this->n = nn;
  }
  int lift(int cur, int k) {
    int new_depth = max(dep[cur] - k, 0);
    while (dep[cur] > new_depth) {
      if (dep[jmp[cur]] >= new_depth)
        cur = jmp[cur];
      else
        cur = fa[cur];
    }
    return cur;
  }
  int lca(int u, int v) {
    if (dep[u] > dep[v])
      swap(u, v);
    v = lift(v, dep[v] - dep[u]);
    while (u != v) {
      if (jmp[v] != jmp[u]) {
        u = jmp[u];
        v = jmp[v];
      } else {
        u = fa[u];
```

```
      v = fa[v];
     }
    }
    return u;
  }
  int dist(int u, int v) { return dep[u] + dep[v] - 2 *
      dep[lca(u, v)]; }
};
```