Team Notebook

January 26, 2023

Contents

1 -Starters-

1.1 C++ Include GNU PBDS [NK]

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
namespace pbds = __gnu_pbds;

template <class T>
using ordered_set = pbds::tree<T, pbds::null_type, std::less
    <T>,
                           pbds::rb_tree_tag,
                           pbds::
                               tree_order_statistics_node_update
                               >;
template <class K, class V>
using hash_map = pbds::gp_hash_table<K, V>;
```

1.2 C++ Starter debug[MB]

```
#include <bits/stdc++.h>

using namespace std;

template <typename T, typename C = typename T::value_type>
typename enable_if<!is_same<T, string>::value, ostream &>::
    type operator<<(ostream &out, const T &c)
{
 for (auto it = c.begin(); it != c.end(); it++)
  out << (it == c.begin() ? "{" : ",") << *it;
 return out << (c.empty() ? "{" : "") << "}";
}

template <typename T, typename S>
ostream &operator<<(ostream &out, const pair<T, S> &p)
{
 return out << "{" << p.first << ", " << p.second << "}";
}

#define dbg(...) _dbg_print(#__VA_ARGS__, __VA_ARGS__);

template <typename Arg1>
void _dbg_print(const char *name, Arg1 &&arg1)
{
 if (name[0] == ' ')
  name++;
 cout << "[" << name << ": " << arg1 << "]"
  << "\n";
}

template <typename Arg1, typename... Args>
void _dbg_print(const char *names, Arg1 &&arg1, Args &&...
    args)
{
 const char *comma = strchr(names + 1, ',');
 cout << "[";
 cout.write(names, comma - names) << ": " << arg1 << "] ";
 _dbg_print(comma + 1, args...);
}
```

1.3 C++ Starter [MB]

```
#if defined LOCAL && !defined ONLINE_JUDGE
#include "debug.cpp"
#else
#include <bits/stdc++.h>
using namespace std;
#define dbg(...) ;
#endif

typedef long long ll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;

#define mem(x, n) memset(x, n, sizeof(x))
#define all(x) x.begin(), x.end()
#define sz(x) ((int)(x).size())
#define vec vector

inline bool read(auto &...a) { return (((cin >> a) ? true :
    false) && ...); }

inline void print(const auto &...a) { ((cout << a), ...); }
inline void println(const auto &...a) { print(a..., '\n'); }

void run_case([[maybe_unused]] const int &TC)
{

}

int main()
{
 ios_base::sync_with_stdio(false), cin.tie(0);

 int tt = 1;
 read(tt);
 for (int tc = 1; tc <= tt; tc++)
  run_case(tc);

 return 0;
}
```

1.4 C++ Starter [NK]

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

constexpr double eps = 1e-9;
constexpr int inf = 1 << 30;
constexpr int mod = 1e9 + 7;
constexpr int nmax = 1e6;

void runcase(int casen) {

    // cout << "Case " << casen << ": " << '\n';
}

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

    int ncases = 1;
    cin >> ncases; // Comment out for single-case tests
    for (int casen = 1; casen <= ncases; ++casen) {
        runcase(casen);
    }

    return 0;
}
```

1.5 Unordered Map [MB]

```
#include <bits/stdc++.h>

// For gp_hash_table
#include <ext/pb_ds/assoc_container.hpp>

using namespace __gnu_pbds;

using namespace std;

struct custom_hash
{
```

```
 static uint64_t splitmix64(uint64_t x)
 {
  // http://xorshift.di.unimi.it/splitmix64.c
  x += 0x9e3779b97f4a7c15;
  x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
  x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
  return x ^ (x >> 31);
 }

 size_t operator()(uint64_t x) const
 {
  static const uint64_t FIXED_RANDOM = chrono::steady_clock
      ::now().time_since_epoch().count();
  return splitmix64(x + FIXED_RANDOM);
 }
};

// Example Use
unordered_map<int, int, custom_hash> mp;

// Faster
gp_hash_table<int, int, custom_hash> mp;
```

2 Brute-force

2.1 Power Set [NK]

```
template <class T>
vector<vector<T>> power_set(const vector<T>& vec) {
    vector<vector<T>> res;
    list<T> buf;
    function<void(int)> recurse = [&](int i) -> void {
        if (i == vec.size()) {
            res.emplace_back(buf.begin(), buf.end());
            return;
        }
        recurse(i + 1);
        buf.push_back(vec[i]), recurse(i + 1), buf.pop_back()
            ;
    };
    recurse(0);
    return res;
}
```

3 Data Structures

3.1 2D Prefix Sum [SA]

```
const int N = 1000, M = 500;
int a[N + 1][M + 1], pref[N + 1][M + 1];

// 1-based
void build() {
    for (int i = 1; i <= N; ++i) {
        for (int j = 1; j <= M; ++j) {
            pref[i][j] = pref[i - 1][j] + pref[i][j - 1] -
                 pref[i - 1][j - 1] + a[i][j];
        }
    }
}
// top_left(i, j), right_bottom(k, l)
auto query(int i, int j, int k, int l) {
    return pref[k][l] - pref[i - 1][l] - pref[k][j - 1] +
         pref[i - 1][j - 1];
}
```

3.2 Articulation Points in O(N + M) [NK]

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}

void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i);
    }
}
```

3.3 Bigint (string) operations [NK[

```
string add(const string& a, const string& b) {
    string sum;
    int i = a.length() - 1, j = b.length() - 1, carry = 0;
    while (i >= 0 || j >= 0) {
        int temp = carry;
        if (i >= 0) {
            temp += (int)(a[i--] - '0');
        }
        if (j >= 0) {
            temp += (int)(b[j--] - '0');
        }
        carry = temp / 10;
        sum += (char)((temp % 10) + '0');
    }
    if (carry > 0) {
        sum += (char)(carry + '0');
    }
    for (int k = sum.length() - 1; k > 0 && sum[k] == '0'; k
         --) {
        sum.pop_back();
    }
    reverse(sum.begin(), sum.end());
    return sum;
}

string multiply(const string& a, const string& b) {
    if (a.length() == 0 || b.length() == 0) {
        return "0";
    }
    string prod = "0";
    int shift = 0, carry = 0;
    for (int j = b.length() - 1; j >= 0; j--) {
        string prod_temp;
```

```
        for (int i = 0; i < shift; i++) {
            prod_temp += '0';
        }
        shift++;
        carry = 0;
        for (int i = a.length() - 1; i >= 0; i--) {
            int temp = ((int)(a[i] - '0') * (int)(b[j] - '0')
                 ) + carry;
            carry = temp / 10;
            prod_temp += (char)((temp % 10) + '0');
        }
        if (carry > 0) {
            prod_temp += (char)(carry + '0');
        }
        reverse(prod_temp.begin(), prod_temp.end());
        prod = add(prod, prod_temp);
    }
    return prod;
}

struct division_t {
    string quot;
    int64_t rem;
};

division_t divide(const string& num, int64_t divisor) {
    string quot;
    int idx = 0;
    int64_t temp = num[idx++] - '0';
    while (temp < divisor && idx < num.length()) {
        temp = (temp * 10) + (int)(num[idx++] - '0');
    }
    quot += (char)((temp / divisor) + '0');
    while (idx < num.length()) {
        temp = ((temp % divisor) * 10) + (int)(num[idx++] - '
            0');
        quot += (char)((temp / divisor) + '0');
    }
    int cnt = 0;
    while (cnt < quot.length() - 1 && quot[cnt] == '0') {
        cnt++;
    }
    quot = quot.substr(cnt);
    return (division_t){quot, temp % divisor};
}
```

3.4 BIT [MB]

```
struct BIT
{
private:
 std::vector<long long> mArray;

public:
 BIT(int sz) // Max size of the array
 {
  mArray.resize(sz + 1, 0);
 }

 void build(const std::vector<long long> &list)
 {
  for (int i = 1; i <= list.size(); i++)
  {
   mArray[i] = list[i];
  }

  for (int ind = 1; ind <= mArray.size(); ind++)
  {
   int ind2 = ind + (ind & -ind);
   if (ind2 <= mArray.size())
   {
    mArray[ind2] += mArray[ind];
   }
  }
 }

 long long prefix_query(int ind)
 {
  int res = 0;
  for (; ind > 0; ind -= (ind & -ind))
  {
   res += mArray[ind];
  }
  return res;
 }

 long long range_query(int from, int to)
 {
  return prefix_query(to) - prefix_query(from - 1);
 }

 void add(int ind, long long add)
 {
  for (; ind < mArray.size(); ind += (ind & -ind))
  {
   mArray[ind] += add;
  }
 }
};
```

3.5 Bridges in O(N + M) [NK]

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

3.6 Bridges Online [NK]

```
vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size;
int bridges;
int lca_iteration;
vector<int> last_visit;

void init(int n) {
    par.resize(n);
    dsu_2ecc.resize(n);
    dsu_cc.resize(n);
    dsu_cc_size.resize(n);
    lca_iteration = 0;
```

```
    last_visit.assign(n, 0);
    for (int i=0; i<n; ++i) {
        dsu_2ecc[i] = i;
        dsu_cc[i] = i;
        dsu_cc_size[i] = 1;
        par[i] = -1;
    }
    bridges = 0;
}

int find_2ecc(int v) {
    if (v == -1)
        return -1;
    return dsu_2ecc[v] == v ? v : dsu_2ecc[v] = find_2ecc(
        dsu_2ecc[v]);
}

int find_cc(int v) {
    v = find_2ecc(v);
    return dsu_cc[v] == v ? v : dsu_cc[v] = find_cc(dsu_cc[v
        ]);
}

void make_root(int v) {
    v = find_2ecc(v);
    int root = v;
    int child = -1;
    while (v != -1) {
        int p = find_2ecc(par[v]);
        par[v] = child;
        dsu_cc[v] = root;
        child = v;
        v = p;
    }
    dsu_cc_size[root] = dsu_cc_size[child];
}

void merge_path (int a, int b) {
    ++lca_iteration;
    vector<int> path_a, path_b;
    int lca = -1;
    while (lca == -1) {
        if (a != -1) {
            a = find_2ecc(a);
            path_a.push_back(a);
            if (last_visit[a] == lca_iteration){
                lca = a;
                break;
                }
            last_visit[a] = lca_iteration;
            a = par[a];
        }
        if (b != -1) {
            b = find_2ecc(b);
            path_b.push_back(b);
            if (last_visit[b] == lca_iteration){
                lca = b;
                break;
                }
            last_visit[b] = lca_iteration;
            b = par[b];
        }

    }

    for (int v : path_a) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
    for (int v : path_b) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
}

void add_edge(int a, int b) {
    a = find_2ecc(a);
    b = find_2ecc(b);
    if (a == b)
        return;

    int ca = find_cc(a);
    int cb = find_cc(b);

    if (ca != cb) {
        ++bridges;
        if (dsu_cc_size[ca] > dsu_cc_size[cb]) {
            swap(a, b);
            swap(ca, cb);
        }
        make_root(a);
        par[a] = dsu_cc[a] = b;
        dsu_cc_size[cb] += dsu_cc_size[a];
    } else {
        merge_path(a, b);
    }
}
```

3.7 Disjoint Set Union [SA]

```
const int N = 100001;
int parent[N], comp_size[N];
int components = 0;

void make_set(int u) {
    parent[u] = u;
    comp_size[u] = 1;
    ++components;
}

int get_size(int u) {
    return comp_size[find(u)];
}

int find(int u) {
    if (u == parent[u]) return u;
    return parent[u] = find(parent[u]);
}

void unite(int u, int v) {
    u = find(u), v = find(v);
    if (u != v) {
        if (comp_size[u] < comp_size[v]) {
            swap(u, v);
        }
        parent[v] = u;
        comp_size[u] += comp_size[v];
        --components;
    }
}
```

3.8 DSU [MB]

```
#include <bits/stdc++.h>

// 0 based
class DSU
{
 std::vector<int> p, csz;

public:
 DSU() {}
 // Max size
```

```
DSU(int dsz)
{
 //Default empty
 p.resize(dsz + 5, 0), csz.resize(dsz + 5, 0);

 init(dsz);
}

void init(int n)
{
 // n = size
 for (int i = 0; i <= n; i++)
 {
  p[i] = i, csz[i] = 1;
 }
}

//Return parent Recursively
int get(int x)
{
 if (p[x] != x)
  p[x] = get(p[x]);

 return p[x];
}

// Return Size
int get_comp_size(int component) { return csz[get(component
    )]; }
// Return if Union created Succesffully or false if they
    are already in Union
bool merge(int x, int y)
{
 x = get(x), y = get(y);
 if (x == y)
  return false;

 if (csz[x] > csz[y])
  std::swap(x, y);

 p[x] = y;
 csz[y] += csz[x];

 return true;
}
};
```

3.9 DSU [NK]

```
struct DSU {
    int n_nodes = 0;
    int n_components = 0;
    vector<int> component_size;
    vector<int> component_root;

    DSU(int n_nodes, bool make_all_nodes = false)
        : n_nodes(n_nodes),
          component_root(n_nodes, -1),
          component_size(n_nodes, 0) {
        if (make_all_nodes) {
            for (int i = 0; i < n_nodes; ++i) {
                make_node(i);
            }
        }
    }

    void make_node(int v) {
        if (component_root[v] == -1) {
            component_root[v] = v;
            component_size[v] = 1;
            ++n_components;
        }
    }

    int root(int v) {
        auto res = v;
        while (component_root[res] != res) {
            res = component_root[res];
        }
        while (v != res) {
            auto u = component_root[v];
            component_root[v] = res;
            v = u;
        }
        return res;
    }

    int connect(int u, int v) {
        u = root(u), v = root(v);
        if (u == v) return u;
        if (component_size[u] < component_size[v]) {
            swap(u, v);
        }
        component_root[v] = u;
        component_size[u] += component_size[v];
        --n_components;
    }
};
```

3.10 Lazy Segment Tree [MB]

```
template <typename T, typename F, T(*op)(T, T), F(*
    lazy_to_lazy)(F, F), T(*lazy_to_seg)(T, F, int, int)>
struct LazySegTree
{
private:
 std::vector<T> segt;
 std::vector<F> lazy;
 int n;
 T neutral;
 F lazyE;
 int left(int si) { return si * 2; }
 int right(int si) { return si * 2 + 1; }
 int midpoint(int ss, int se) { return (ss + (se - ss) / 2);
     }
 T query(int ss, int se, int si, int qs, int qe)
 {
  // **** //
  if (lazy[si] != lazyE)
  {
   F curr = lazy[si];
   lazy[si] = lazyE;
   segt[si] = lazy_to_seg(segt[si], curr, ss, se);
   if (ss != se)
   {
    lazy[left(si)] = lazy_to_lazy(lazy[left(si)], curr);
    lazy[right(si)] = lazy_to_lazy(lazy[right(si)], curr);
   }
  }
  if (se < qs || qe < ss)
   return neutral;
  if (qs <= ss && qe >= se)
   return segt[si];
  int mid = midpoint(ss, se);
  return op(query(ss, mid, left(si), qs, qe), query(mid + 1,
      se, right(si), qs, qe));
 }

 void update(int ss, int se, int si, int qs, int qe, F val)
 {
  // **** //
  if (lazy[si] != lazyE)
  {
   F curr = lazy[si];
   lazy[si] = lazyE;
   segt[si] = lazy_to_seg(segt[si], curr, ss, se);
   if (ss != se)
   {
    lazy[left(si)] = lazy_to_lazy(lazy[left(si)], curr);
```

```
   lazy[right(si)] = lazy_to_lazy(lazy[right(si)], curr);
  }
 }
 if (se < qs || qe < ss)
  return;
 if (qs <= ss && qe >= se)
 {
  // **** //
  segt[si] = lazy_to_seg(segt[si], val, ss, se);

  if (ss != se)
  {
   lazy[left(si)] = lazy_to_lazy(lazy[left(si)], val);
   lazy[right(si)] = lazy_to_lazy(lazy[right(si)], val);
  }
  return;
 }

 int mid = midpoint(ss, se);

 update(mid + 1, se, si * 2 + 1, qs, qe, val);
 update(ss, mid, left(si), qs, qe, val);

 segt[si] = op(segt[left(si)], segt[right(si)]);
}
void build(const std::vector<T> &a, int si, int ss, int se)
{
 if (ss == se)
 {
  segt[si] = a[ss];
  return;
 }
 int mid = midpoint(ss, se);
 build(a, left(si), ss, mid);
 build(a, right(si), mid + 1, se);
 segt[si] = op(segt[left(si)], segt[right(si)]);
}
public:
 LazySegTree() : n(0) {}
 LazySegTree(int sz, T ini, T _neutral, F _lazyE)
 {
  this->n = sz + 1;
  this->neutral = _neutral;
  this->lazyE = _lazyE;
  segt.resize(n * 4 + 5, ini);
  lazy.resize(n * 4 + 5, _lazyE);
 }
 LazySegTree(const std::vector<T> &arr, T ini, T _neutral, F
      _lazyE) : LazySegTree((int)arr.size(), ini, _neutral,
     _lazyE)
 {
  init(arr);
 }
 void init(const std::vector<T> &arr) { this->n = (int)arr.
     size(); build(arr, 1, 0, n - 1); }
 T get(int qs, int qe) { return query(0, n - 1, 1, qs, qe);
     }
 void set(int from, int to, F val) { update(0, n - 1, 1,
     from, to, val); }
};

int op(int a, int b)
{
 return a + b;
}

int lazy_to_seg(int seg, int lazy_v, int l, int r)
{
 return seg + (lazy_v * (r - l + 1));
}

int lazy_to_lazy(int curr_lazy, int input_lazy)
{
 return curr_lazy + input_lazy;
}
```

3.11 Lazy Segment Tree [NK

```
/**
 * @brief Segment tree with lazy updates.
 * @tparam ValueTp The value type. Must imply a monoid
 * (i.e., have a closed, associative binary operation and a
     corresponding identity
 * element).
 * @tparam FnCombine A function to combine two values into
     one. Implements the closed,
 * associative binary operation of the ValueTp monoid.
 * @tparam FnGetDefaultValue A function that returns the
     default value for any node.
 * The returning value is the identity element of the
     ValueTp monoid.
 * @tparam ArgTp The type of lazy-update arguments. Must
     imply a monoid
 * (i.e., have a closed, associative binary operation and a
     corresponding identity
 * element).
 * @tparam FnCompose A function to compose two lazy-update
     arguments into one.
 * Implements the closed, associative binary operation of
     the ArgTp monoid.
 * @tparam FnGetDefaultArg A function that returns the
     default lazy-update argument
 * for any node. The returning value is the identity element
      of the ArgTp monoid.
 * @tparam FnApply A function to apply an update on a node's
      value. Takes the
 * following parameters: the node's value, an update
     argument, and two indexes
 * indicating the range of the segment covered by the node.
 */
template <class ValueTp,
          ValueTp (*FnCombine)(ValueTp, ValueTp),
          ValueTp (*FnGetDefaultValue)(),
          class ArgTp,
          ArgTp (*FnCompose)(ArgTp, ArgTp),
          ArgTp (*FnGetDefaultArg)(),
          ValueTp (*FnApply)(ValueTp, ArgTp, std::size_t, std
              ::size_t)>
class Lazy_segment_tree {
public:
    using SizeType = std::size_t;
    using ValueType = ValueTp;
    using ArgType = ArgTp;

    static constexpr auto combine = FnCombine;
    static constexpr auto default_value = FnGetDefaultValue;
    static constexpr auto compose = FnCompose;
    static constexpr auto default_arg = FnGetDefaultArg;
    static constexpr auto apply = FnApply;

    /**
     * @brief Default constructor.
     */
    Lazy_segment_tree() {}

    /**
     * @brief Constructs and builds a tree over a default-
         valued array.
     * @param n Size of the array
     */
    Lazy_segment_tree(SizeType n) { build(n); }

    /**
     * @brief Constructs and builds a tree over a range of
         values.
     * @tparam InputIterator An input iterator type
     * @param from Iterator pointing to the beginning of the
         range
```

```
 * @param until Iterator pointing to the end (one place
     past the last) of the range
 */
template <class InputIterator>
Lazy_segment_tree(InputIterator from, InputIterator until
    ) { build(from, until); }

/**
 * @brief Builds the tree over a default-valued array.
 * @param n Size of the array
 */
void build(SizeType n) {
    log2_n_ = 0;
    while (((SizeType)1 << log2_n_) < n) ++log2_n_;
    n_ = 1 << log2_n_;
    ranges_.resize(n_ << 1);
    for (SizeType i = n_; i < (n_ << 1); ++i) {
        ranges_[i][0] = i - n_, ranges_[i][1] = ranges_[i
            ][0] + 1;
    }
    for (SizeType i = n_ - 1; i; --i) {
        ranges_[i][0] = std::min(ranges_[i << 1][0],
            ranges_[i << 1 | 1][0]);
        ranges_[i][1] = std::max(ranges_[i << 1][1],
            ranges_[i << 1 | 1][1]);
    }
    tree_.assign(n_ << 1, default_value());
    args_.assign(n_, default_arg());
}

/**
 * @brief Builds the tree over a range of values.
 * @tparam InputIterator An input iterator type
 * @param from Iterator pointing to the beginning of the
     range
 * @param until Iterator pointing to the end (one past
     the last element) of the range
 */
template <class InputIterator>
void build(InputIterator from, InputIterator until) {
    const std::vector<ValueType> v(from, until);
    build(v.size());
    for (SizeType i = 0; i < v.size(); ++i) {
        tree_[i + n_] = v[i];
    }
    for (SizeType i = n_ - 1; i; --i) {
        tree_[i] = combine(tree_[i << 1], tree_[i << 1 |
            1]);
    }
}

/**
 * @brief Performs a point-update (update at a single
     position) on the segment tree.
 * @param p Index of the element to update
 * @param arg Argument of the update
 */
void update(SizeType p, ArgType arg) {
    assert(0 <= p && p < n_);
    apply_update(p + n_, arg);
    build_update(p + n_);
}

/**
 * @brief Performs a lazy range-update on the segment
     tree.
 * @param l Index pointing to the begining of the range
 * @param r Index pointing to the end (one past the last
     element) of the range
 * @param arg Argument of the update
 */
void update(SizeType l, SizeType r, ArgType arg) {
    assert(0 <= l && l <= r && r <= n_);
    l += n_, r += n_;
    const auto l0 = l, r0 = r;
    propagate_update(l0), propagate_update(r0 - 1);
    while (l < r) {
        if (l & 1) {
            apply_update(l++, arg);
        }
        if (r & 1) {
            apply_update(--r, arg);
        }
        l >>= 1, r >>= 1;
    }
    build_update(l0), build_update(r0 - 1);
}

/**
 * @brief Returns the value of a segment.
 * @param l Index pointing to the begining of the range
 * @param r Index pointing to the end (one past the last
     element) of the range
 * @return ValueType
 */
ValueType operator()(SizeType l, SizeType r) {
    assert(0 <= l && l <= r && r <= n_);
    ValueType result = default_value();
    l += n_, r += n_;
    propagate_update(l), propagate_update(r - 1);
    while (l < r) {
        if (l & 1) {
            result = combine(result, tree_[l++]);
        }
        if (r & 1) {
            result = combine(result, tree_[--r]);
        }
        l >>= 1, r >>= 1;
    }
    return result;
}

/**
 * @brief Returns the segment tree.
 * @return std::vector<ValueType>
 */
std::vector<ValueType> tree() const { return tree_; }

/**
 * @brief Returns the lazy-update arguments.
 * @return std::vector<ArgType>
 */
std::vector<ArgType> args() const { return args_; }

/**
 * @brief Returns the ranges covered by tree nodes. Each
     range is an
 * array of two indexes, the first one being the
     beginning, the second
 * one being the end (one past the last index).
 * @return std::vector<std::array<SizeType, 2>>
 */
std::vector<std::array<SizeType, 2>> ranges() const {
    return ranges_; }

private:
    SizeType n_ = 0;
    int log2_n_ = 0;
    std::vector<ValueTp> tree_;
    std::vector<ArgTp> args_;
    std::vector<std::array<SizeType, 2>> ranges_;

    void apply_update(SizeType i, const ArgTp& arg) {
        tree_[i] = apply(tree_[i], arg, ranges_[i][0],
            ranges_[i][1]);
        if (i < n_) args_[i] = compose(args_[i], arg);
    }

    void propagate_update(SizeType i) {
        assert(n_ <= i && i < (n_ << 1));
```

```
        for (int h = log2_n_; h; --h) {
            auto j = (i >> h);
            apply_update(j << 1, args_[j]);
            apply_update(j << 1 | 1, args_[j]);
            args_[j] = default_arg();
        }
    }

    void build_update(SizeType i) {
        assert(n_ <= i && i < (n_ << 1));
        while (i >>= 1) {
            tree_[i] = apply(combine(tree_[i << 1], tree_[i
                << 1 | 1]),
                        args_[i],
                        ranges_[i][0],
                        ranges_[i][1]);
        }
    }
};

using Val_t = int64_t;
constexpr Val_t combine(Val_t x, Val_t y) { return x + y; }
constexpr Val_t defval() { return 0; }

using Arg_t = int64_t;
constexpr Arg_t compose(Arg_t p, Arg_t q) { return p + q; }
constexpr Arg_t defarg() { return 0; }

constexpr Val_t apply(Val_t val, Arg_t arg, size_t l, size_t
    r) {
    return val + (arg * (r - l));
}

using Segtree =
    Lazy_segment_tree<Val_t, combine, defval, Arg_t, compose,
        defarg, apply>;
```

3.12 LCA [MB]

```
struct LCA
{
private:
 int n, lg;
 std::vector<int> depth;
 std::vector<std::vector<int>> up;
 std::vector<std::vector<int>> g;

public:
 LCA() : n(0), lg(0) {}

LCA(int _n)
{
 this->n = _n;
 lg = (int)log2(n) + 2;
 depth.resize(n + 5, 0);
 up.resize(n + 5, std::vector<int>(lg, 0));
 g.resize(n + 1);
}

LCA(std::vector<std::vector<int>> &graph) : LCA((int)graph.
    size())
{
 for (int i = 0; i < (int)graph.size(); i++)
  g[i] = graph[i];

 dfs(1, 0);
}

void dfs(int curr, int p)
{
 up[curr][0] = p;
 for (int next : g[curr])
 {
  if (next == p)
   continue;
  depth[next] = depth[curr] + 1;
  up[next][0] = curr;
  for (int j = 1; j < lg; j++)
   up[next][j] = up[up[next][j - 1]][j - 1];
  dfs(next, curr);
 }
}

void clear_v(int a)
{
 g[a].clear();
}

void clear(int n_ = -1)
{
 if (n_ == -1)
  n_ = ((int)(g.size())) - 1;

 for (int i = 0; i <= n_; i++)
 {
  g[i].clear();
 }
}

void add(int a, int b)
{
 g[a].push_back(b);
}

int par(int a)
{
 return up[a][0];
}

int get_lca(int a, int b)
{
 if (depth[a] < depth[b])
  std::swap(a, b);

 int k = depth[a] - depth[b];
 for (int j = lg - 1; j >= 0; j--)
 {
  if (k & (1 << j))
   a = up[a][j];
 }

 if (a == b)
  return a;

 for (int j = lg - 1; j >= 0; j--)
  if (up[a][j] != up[b][j])
  {
   a = up[a][j];
   b = up[b][j];
  }

 return up[a][0];
}

int get_dist(int a, int b)
{
 return depth[a] + depth[b] - 2 * depth[get_lca(a, b)];
}
};
```

3.13 Lowest Common Ancestor [SA]

```
vector<int> dist;
vector<vector<int>> up;
vector<vector<int>> adj;
int lg = -1;

void dfs(int u, int p = -1) {
```

```
    up[u][0] = p;
    for (auto v : adj[u]) {
        if (dist[v] != -1) continue;
        dist[v] = 1 + dist[u];
        dfs(v, u);
    }
}

void pre_process(int root, int n) {
    assert(lg != -1);
    dist[root] = 0;
    dfs(root);
    for (int i = 1; i < lg; ++i) {
        for (int j = 1; j <= n; ++j) {// 1-based graph
            int p = up[j][i - 1];
            if (p == -1) continue;
            up[j][i] = up[p][i - 1];
        }
    }
}

int get_lca(int u, int v) {
    if (dist[u] > dist[v])
        swap(u, v);

    int dif = dist[v] - dist[u];
    while (dif > 0) {
        int lg = __lg(dif);
        v = up[v][lg];
        dif -= (1 << lg);
    }
    if (u == v)
        return u;

    for (int i = lg - 1; i >= 0; --i) {
        if (up[u][i] == up[v][i]) continue;
        u = up[u][i];
        v = up[v][i];
    }
    return up[u][0];
}

int get_kth_ancestor(int v, int k) {
    while (k > 0) {
        int lg = __lg(k);
        v = up[v][lg];
        k -= (1 << lg);
    }
    return v;
}
```

3.14 Mos Algorithm [MB]

```
#include <bits/stdc++.h>

using namespace std;

const int N = 3e4 + 5;
const int blck = sqrt(N) + 1;

struct Query
{
 int l, r, i;
 bool operator<(const Query q) const
 {
  if (this->l / blck == q.l / blck)
   return this->r < q.r;
  return this->l / blck < q.l / blck;
 }
};

vector<int> mos_alogorithm(vector<Query> &queries, vector<
    int> &a)
{
 vector<int> answers(queries.size());
 sort(queries.begin(), queries.end());

 int sza = 1e6 + 5;
 vector<int> freq(sza);

 int cnt = 0;

 auto add = [&](int x) -> void
 {
  freq[x]++;
  if (freq[x] == 1)
   cnt++;
 };

 auto remove = [&](int x) -> void
 {
  freq[x]--;
  if (freq[x] == 0)
   cnt--;
 };

 int l = 0;
 int r = -1;
 for (Query q : queries)
 {
  while (l > q.l)
  {
   l--;
   add(a[l]);
  }
  while (r < q.r)
  {
   r++;
   add(a[r]);
  }
  while (l < q.l)
  {
   remove(a[l]);
   l++;
  }
  while (r > q.r)
  {
   remove(a[r]);
   r--;
  }
  answers[q.i] = cnt;
 }
 return answers;
}

int main()
{
 int n;
 cin >> n;

 vector<int> a(n);
 for (int i = 0; i < n; i++)
  cin >> a[i];

 int q;
 cin >> q;

 vector<Query> qr(q);

 for (int i = 0; i < q; i++)
 {
  int l, r;
  cin >> l >> r;

  l--, r--;
  qr[i].l = l, qr[i].r = r, qr[i].i = i;
 }

 vector<int> res = mos_alogorithm(qr, a);

 for (int i = 0; i < q; i++)
```

```
  cout << res[i] << endl;

 return 0;
}
```

3.15 SCC, Condens Graph [NK]

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

vector<vector<int>> adj, adj_rev;
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
    used[v] = true;

    for (auto u : adj[v])
        if (!used[u])
            dfs1(u);

    order.push_back(v);
}

void dfs2(int v) {
    used[v] = true;
    component.push_back(v);

    for (auto u : adj_rev[v])
        if (!used[u])
            dfs2(u);
}

int main() {
    int n;
    // ... read n ...

    for (;;) {
        int a, b;
        // ... read next directed edge (a,b) ...
        adj[a].push_back(b);
        adj_rev[b].push_back(a);
    }

    used.assign(n, false);

    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i);

    used.assign(n, false);
    reverse(order.begin(), order.end());

    for (auto v : order)
        if (!used[v]) {
            dfs2(v);

            // ... processing next component ...

            component.clear();
        }

    vector<int> roots(n, 0);
    vector<int> root_nodes;
    vector<vector<int>> adj_scc(n);

    for (auto v : order)
        if (!used[v]) {
            dfs2(v);

            int root = component.front();
            for (auto u : component) roots[u] = root;
            root_nodes.push_back(root);

            component.clear();
        }

    for (int v = 0; v < n; v++)
        for (auto u : adj[v]) {
            int root_v = roots[v],
                root_u = roots[u];

            if (root_u != root_v)
                adj_scc[root_v].push_back(root_u);
        }
}
```

3.16 Segment Tree[MB]

```
template <typename T, T(*op)(T, T)>
struct SegTree
{
private:
 std::vector<T> segt;
 int n;
 T e;
 int left(int si) { return si * 2; }
 int right(int si) { return si * 2 + 1; }
 int midpoint(int ss, int se) { return (ss + (se - ss) / 2);
      }
 T query(int ss, int se, int qs, int qe, int si)
 {
  if (se < qs || qe < ss)
   return e;
  if (qs <= ss && qe >= se)
   return segt[si];
  int mid = midpoint(ss, se);
  return op(query(ss, mid, qs, qe, left(si)), query(mid + 1,
       se, qs, qe, right(si)));
 }
 void update(int ss, int se, int key, int si, T val)
 {
  if (ss == se)
  {
   segt[si] = val;
   return;
  }
  int mid = midpoint(ss, se);
  if (key > mid)
   update(mid + 1, se, key, right(si), val);
  else
   update(ss, mid, key, left(si), val);
  segt[si] = op(segt[left(si)], segt[right(si)]);
 }
 void build(const std::vector<T> &a, int si, int ss, int se)
 {
  if (ss == se)
  {
   segt[si] = a[ss];
   return;
  }
  int mid = midpoint(ss, se);
  build(a, left(si), ss, mid);
  build(a, right(si), mid + 1, se);
  segt[si] = op(segt[left(si)], segt[right(si)]);
 }
public:
 SegTree() : n(0) {}
 SegTree(int sz, T _e)
 {
  this->e = _e;
  this->n = sz;
  segt.resize(n * 4 + 5, _e);
 }
 SegTree(const std::vector<T> &arr, T _e) : SegTree((int)arr
      .size(), _e) { init(arr); }
 void init(const std::vector<T> &arr) { this->n = (int)(arr.
      size());build(arr, 1, 0, n - 1); }
```

```
 T get(int qs, int qe) { return query(0, n - 1, qs, qe, 1);
     }
 void set(int key, T val) { update(0, n - 1, key, 1, val); }
};

int op(int a, int b)
{
 return min(a, b);
}
```

3.17 Sparse Table [SA]

```
const int N = 100001, LG = 18;
int st[N][LG];

void sparse_table(vector<int>& a, int n) {
    for (int i = 0; i < n; ++i) {
        st[i][0] = a[i];
    }

    for (int j = 1; j < LG; ++j) {
        for (int i = 0; i + (1 << j) - 1 < n; ++i) {
            st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1)
                )][j - 1]);
        }
    }
}

int rmq(int L, int R) {
    int lg = __lg(R - L + 1);
    return min(st[L][lg], st[R - (1 << lg) + 1][lg]);
}
```

3.18 SparseTable[MB]

```
template <typename T, T (*op)(T, T)>
struct SparseTable
{
private:
 std::vector<std::vector<T>> st;
 int n, lg;
 std::vector<int> logs;
 T e;

public:
 SparseTable() : n(0) {}
 SparseTable(int _n)
 {
  this->n = _n;
  int bit = 0;
  while ((1 << bit) <= n)
   bit++;
  this->lg = bit;

  st.resize(n, std::vector<T>(lg));
  logs.resize(n + 1, 0);
  logs[1] = 0;
  for (int i = 2; i <= n; i++)
  {
   logs[i] = logs[i / 2] + 1;
  }
 }

 SparseTable(const std::vector<T> &a) : SparseTable((int)a.
     size())
 {
  init(a);
 }

 void init(const std::vector<T> &a)
 {
  this->n = (int)a.size();

  for (int i = 0; i < n; i++)
  {
   st[i][0] = a[i];
  }

  for (int j = 1; j <= lg; j++)
  {
   for (int i = 0; i + (1 << j) <= n; i++)
   {
    st[i][j] = op(st[i][j - 1], st[std::min(i + (1 << (j - 1)
        ), n - 1)][j - 1]);
   }
  }
 }

 T get(int l, int r)
 {
  int j = logs[r - l + 1];
  return op(st[l][j], st[r - (1 << j) + 1][j]);
 }
};

int min(int a, int b)
{
 return std::min(a, b);
}
```

3.19 Treap[MB]

```
#include <bits/stdc++.h>

#define mem(x, n) memset(x, n, sizeof(x))
#define all(x) x.begin(), x.end()
#define endl "\n"

#include <ext/pb_ds/assoc_container.hpp> // Common file

// using namespace __gnu_pbds;

// https://codeforces.com/blog/entry/11080
//cout<<*X.find_by_order(4)<<endl; // 16
// cout<<(end(X)==X.find_by_order(6))<<endl; // true
// cout<<X.order_of_key(-5)<<endl; // 0
template <typename T, typename order = std::less<T>>
using ordered_set = __gnu_pbds::tree<T, __gnu_pbds::
    null_type, order, __gnu_pbds::rb_tree_tag, __gnu_pbds::
    tree_order_statistics_node_update>;

int main()
{
 ordered_set<int> X;

 std::cout << *X.find_by_order(4) << endl;          // 16
 std::cout << (std::end(X) == X.find_by_order(6)) << endl;
     // true
 std::cout << X.order_of_key(-5) << endl;          // 0

 return 0;
}
```

4 Graph

4.1 Edge Remove CC [MB]

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
```

```
#define var(...) " [" << #__VA_ARGS__ ": " << (__VA_ARGS__)
    << "] "
#define mem(x, n) memset(x, n, sizeof(x))
#define all(x) x.begin(), x.end()
#define sz(x) ((int)x.size())
#define vec vector
#define endl "\n"

class DSU
{
 std::vector<int> p, csz;

public:
 DSU() {}

 DSU(int dsz) // Max size
 {
  //Default empty
  p.resize(dsz + 5, 0), csz.resize(dsz + 5, 0);

  init(dsz);
 }

 void init(int n)
 {
  // n = size
  for (int i = 0; i <= n; i++)
  {
   p[i] = i, csz[i] = 1;
  }
 }

 //Return parent Recursively
 int get(int x)
 {
  if (p[x] != x)
   p[x] = get(p[x]);

  return p[x];
 }

 // Return Size
 int getSize(int x) { return csz[get(x)]; }
 // Return if Union created Succesffully or false if they
     are already in Union
 bool merge(int x, int y)
 {
  x = get(x), y = get(y);
  if (x == y)
   return false;

  if (csz[x] > csz[y])
   std::swap(x, y);

  p[x] = y;
  csz[y] += csz[x];

  return true;
 }
};

void runCase([[maybe_unused]] const int &TC)
{
 int n, m;
 cin >> n >> m;

 auto g = vec(n + 1, set<int>());

 auto dsu = DSU(n + 1);

 for (int i = 0; i < m; i++)
 {
  int u, v;
  cin >> u >> v;

  g[u].insert(v);
  g[v].insert(u);
 }

 set<int> elligible;

 for (int i = 1; i <= n; i++)
 {
  elligible.insert(i);
 }

 int i = 1;
 int cnt = 0;

 while (sz(elligible))
 {
  cnt++;
  queue<int> q;
  q.push(*elligible.begin());
  elligible.erase(elligible.begin());

  while (sz(q))
  {
   int fr = q.front();
   q.pop();

   auto v = elligible.begin();

   while (v != elligible.end())
   {
    if (g[fr].find(*v) == g[fr].end())
    {
     q.push(*v);
     v = elligible.erase(v);
    }
    else
    {
     v++;
    }
   }
  }
 }

 cout << cnt - 1 << endl;
}

int main()
{
 ios_base::sync_with_stdio(false), cin.tie(0);

 int t = 1;
 //cin >> t;

 for (int tc = 1; tc <= t; tc++)
  runCase(tc);

 return 0;
}
```

4.2 Kruskal's [NK]

```
struct Edge {
    using weight_type = long long;
    static const weight_type bad_w; // Indicates non-existent
          edge

    int u = -1;          // Edge source (vertex id)
    int v = -1;          // Edge destination (vertex id)
    weight_type w = bad_w; // Edge weight

#define DEF_EDGE_OP(op)                                        \
    friend bool operator op(const Edge& lhs, const Edge& rhs)
          { \
```

```
        return make_pair(lhs.w, make_pair(lhs.u, lhs.v)) op \
            make_pair(rhs.w, make_pair(rhs.u, rhs.v));     \
    }

    DEF_EDGE_OP(==)
    DEF_EDGE_OP(!=)
    DEF_EDGE_OP(<)
    DEF_EDGE_OP(<=)
    DEF_EDGE_OP(>)
    DEF_EDGE_OP(>=)
};

constexpr Edge::weight_type Edge::bad_w = numeric_limits<
     Edge::weight_type>::max();

template <class EdgeCompare = less<Edge>>
constexpr vector<Edge> kruskal(const int n, vector<Edge>
     edges, EdgeCompare compare = EdgeCompare()) {
    // define dsu part and initlaize forests

    vector<int> parent(n);
    iota(parent.begin(), parent.end(), 0);
    vector<int> size(n, 1);
    auto root = [&](int x) {
        int r = x;
        while (parent[r] != r) {
            r = parent[r];
        }
        while (x != r) {
            int tmp_id = parent[x];
            parent[x] = r;
            x = tmp_id;
        }
        return r;
    };
    auto connect = [&](int u, int v) {
        u = root(u);
        v = root(v);
        if (size[u] > size[v]) {
            swap(u, v);
        }
        parent[v] = u;
        size[u] += size[v];
        size[v] = 0;
    };

    // connect components (trees) with edges in order from
         the sorted list

    sort(edges.begin(), edges.end(), compare);

    vector<Edge> edges_mst;
    int remaining = n - 1;
    for (const Edge& e : edges) {
        if (!remaining) break;
        const int u = root(e.u);
        const int v = root(e.v);
        if (u == v) continue;
        --remaining;
        edges_mst.push_back(e);
        connect(u, v);
    }

    return edges_mst;
}
```

4.3 Tree Rooting [MB]

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

const int N = 2e5 + 5;

vector<int> g[N];
ll sz[N], dist[N], sum[N];

void dfs(int s, int p)
{
 sz[s] = 1;
 dist[s] = 0;
 for (int nxt : g[s])
 {
  if (nxt == p)
   continue;
  dfs(nxt, s);
  sz[s] += sz[nxt];
  dist[s] += (dist[nxt] + sz[nxt]);
 }
}

void dfs1(int s, int p)
{

 if (p != 0)
 {
  ll my_size = sz[s];
  ll my_contrib = (dist[s] + sz[s]);
  sum[s] = sum[p] - my_contrib + sz[1] - sz[s] + dist[s];
 }
 for (int nxt : g[s])
 {
  if (nxt == p)
   continue;
  dfs1(nxt, s);
 }
}

// problem link: https://cses.fi/problemset/task/1133

int main()
{
 int n;
 cin >> n;

 for (int i = 1, u, v; i < n; i++)
  cin >> u >> v, g[u].push_back(v), g[v].push_back(u);

 dfs(1, 0);

 sum[1] = dist[1];

 dfs1(1, 0);

 for (int i = 1; i <= n; i++)
  cout << sum[i] << " ";
 cout << endl;

 return 0;
}
```

5 Math

5.1 Combinatrics [MB]

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

struct Combinatrics
{
 vector<ll> fact, fact_inv, inv;
 ll mod, nl;
```

```
Combinatrics() {}

Combinatrics(ll n, ll _mod)
{
 this->nl = n;
 this->mod = _mod;
 fact.resize(n + 1, 1), fact_inv.resize(n + 1, 1), inv.
     resize(n + 1, 1);
 init();
}

void init()
{
 fact[0] = 1;

 for (int i = 1; i <= nl; i++)
 {
  fact[i] = (fact[i - 1] * i) % mod;
 }

 inv[0] = inv[1] = 1;
 for (int i = 2; i <= nl; i++)
  inv[i] = inv[mod % i] * (mod - mod / i) % mod;

 fact_inv[0] = fact_inv[1] = 1;

 for (int i = 2; i <= nl; i++)
  fact_inv[i] = (inv[i] * fact_inv[i - 1]) % mod;
}

ll ncr(ll n, ll r)
{
 if(n < r){
  return 0;
 }

 if (n > nl)
  return ncr(n, r, mod);
 return (((fact[n] * 1LL * fact_inv[r]) % mod) * 1LL *
     fact_inv[n - r]) % mod;
}

ll npr(ll n, ll r)
{
 if(n < r){
  return 0;
 }

 if (n > nl)
  return npr(n, r, mod);
 return (fact[n] * 1LL * fact_inv[n - r]) % mod;
}

ll big_mod(ll a, ll p, ll m = -1)
{
 m = (m == -1 ? mod : m);
 ll res = 1 % m, x = a % m;
 while (p > 0)
  res = ((p & 1) ? ((res * x) % m) : res), x = ((x * x) % m
      ), p >>= 1;
 return res;
}

ll mod_inv(ll a, ll p)
{
 return big_mod(a, p - 2, p);
}

ll ncr(ll n, ll r, ll p)
{
 if (n < r)
  return 0;
 if (r == 0)
  return 1;
 return (((fact[n] * mod_inv(fact[r], p)) % p) * mod_inv(
     fact[n - r], p)) % p;
}

ll npr(ll n, ll r, ll p)
{
 if (n < r)
  return 0;
 if (r == 0)
  return 1;
 return (fact[n] * mod_inv(fact[n - r], p)) % p;
}
};

const int N = 1e6, MOD = 998244353;

Combinatrics comb(N, MOD);
```

5.2 Extended GCD [NK]

```
template <class Z>
constexpr Z extended_gcd(Z a, Z b, Z& x_ref, Z& y_ref) {
    x_ref = 1, y_ref = 0;
    Z x1 = 0, y1 = 1, tmp = 0, q = 0;
    while (b > 0) {
        q = a / b;
        tmp = a, a = b, b = tmp - (q * b);
        tmp = x_ref, x_ref = x1, x1 = tmp - (q * x1);
        tmp = y_ref, y_ref = y1, y1 = tmp - (q * y1);
    }
    return a;
}
```

5.3 Fraction[MB]

```
struct Fraction {
    int p, q;

    Fraction (int _p, int _q) : p(_p), q(_q) {
    }

    std::strong_ordering operator<=> (const Fraction &oth)
        const {
        return p * oth.q <=> q * oth.p;
    }
};
```

5.4 Mathematical Progression [SA]

```
int arithmetic_nth_term(int a, int n, int d) {
    return a + (n - 1) * d;
}
int arithmetic_sum(int a, int n, int d) {
    return n * (2 * a + (n - 1) * d) / 2;
}
int geometric_nth_term(int a, int n, int r) {
    return a * pow(r, n - 1);
}
int geometric_sum(int a, int n, int r) {
    if (r == 1) return n * a;
    if (r < 1) return a * (1 - pow(r, n)) / (1 - r);
    else return a * (pow(r, n) - 1) / (r - 1);
}
int infinite_geometric_sum(int a, int r) {
    assert(r < 1);
    return a / (1 - r);
}
```

5.5 Miller-Rabin-for-prime-checking [SK]

```
typedef long long ll;

ll mulmod(ll a, ll b, ll c) {
  ll x = 0, y = a % c;
  while (b) {
    if (b & 1) x = (x + y) % c;
    y = (y << 1) % c;
    b >>= 1;
  }
  return x % c;
}

ll fastPow(ll x, ll n, ll MOD) {
  ll ret = 1;
  while (n) {
    if (n & 1) ret = mulmod(ret, x, MOD);
    x = mulmod(x, x, MOD);
    n >>= 1;
  }
  return ret;
}

bool isPrime(ll n) {
  ll d = n - 1;
  int s = 0;
  while (d % 2 == 0) {
    s++;
    d >>= 1;
  }

  // It's guranteed that these values will work for any
       number smaller than 3*10**18 (3 and 18 zeros)
  int a[9] = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
  for(int i = 0; i < 9; i++) {
    bool comp = fastPow(a[i], d, n) != 1;
    if(comp) for(int j = 0; j < s; j++) {
      ll fp = fastPow(a[i], (1LL << (ll)j)*d, n);
      if (fp == n - 1) {
        comp = false;
        break;
      }
    }
    if(comp) return false;
  }
  return true;
}
```

5.6 Modular Binary Exponentiation (Power) [NK]

```
template <class B, class E, class M>
constexpr B power(B base, E expo, M mod = 0) {
    assert(expo >= 0);
    if (mod == 1) return 0;
    if (base == 0 || base == 1) return base;
    B res = 1;
    if (!mod) {
        while (expo) {
            if (expo & 1) res *= base;
            base *= base;
            expo >>= 1;
        }
    } else {
        assert(mod > 0);
        base %= mod;
        if (base <= 1) return base;
        while (expo) {
            if (expo & 1) res = (res * base) % mod;
            base = (base * base) % mod;
            expo >>= 1;
        }
    }
    return res;
}
```

5.7 Modular Int [MB]

```
#include <bits/stdc++.h>
// Tested By Ac
// submission : https://atcoder.jp/contests/abc238/
     submissions/29247261
// problem : https://atcoder.jp/contests/abc238/tasks/
     abc238_c

template <const int MOD>
struct ModInt
{
 int val;
 ModInt() { val = 0; }
 ModInt(long long v) { v += (v < 0 ? MOD : 0), val = (int)(v
      % MOD); }
 ModInt &operator+=(const ModInt &rhs)
 {
  val += rhs.val, val -= (val >= MOD ? MOD : 0);
  return *this;
 }
 ModInt &operator-=(const ModInt &rhs)
 {
  val -= rhs.val, val += (val < 0 ? MOD : 0);
  return *this;
 }
 ModInt &operator*=(const ModInt &rhs)
 {
  val = (int)((val * 1ULL * rhs.val) % MOD);
  return *this;
 }
 ModInt pow(long long n) const
 {
  ModInt x = *this, r = 1;
  while (n)
   r = ((n & 1) ? r * x : r), x = (x * x), n >>= 1;
  return r;
 }
 ModInt inv() const { return this->pow(MOD - 2); }
 ModInt &operator/=(const ModInt &rhs) { return *this = *
      this * rhs.inv(); }
 friend ModInt operator+(const ModInt &lhs, const ModInt &
      rhs) { return ModInt(lhs) += rhs; }
 friend ModInt operator-(const ModInt &lhs, const ModInt &
      rhs) { return ModInt(lhs) -= rhs; }
 friend ModInt operator*(const ModInt &lhs, const ModInt &
      rhs) { return ModInt(lhs) *= rhs; }
 friend ModInt operator/(const ModInt &lhs, const ModInt &
      rhs) { return ModInt(lhs) /= rhs; }
 friend bool operator==(const ModInt &lhs, const ModInt &rhs
      ) { return lhs.val == rhs.val; }
 friend bool operator!=(const ModInt &lhs, const ModInt &rhs
      ) { return lhs.val != rhs.val; }
 friend std::ostream &operator<<(std::ostream &out, const
      ModInt &m) { return out << m.val; }
 friend std::istream &operator>>(std::istream &in, ModInt &m
      ) { return in >> m.val; }
 operator int() const { return val; }
};

const int MOD = 1e9 + 7;
using mint = ModInt<MOD>;
```

5.8 Modular inverse [NK]

```
template <class Z>
constexpr Z inverse(Z num, Z mod) {
    assert(mod > 1);
    if (!(0 <= num && num < mod)) {
```

```
        num %= mod;
        if (num < 0) num += mod;
    }
    Z res = 1, tmp = 0;
    assert(extended_gcd(num, mod, res, tmp) == 1);
    if (res < 0) res += mod;
    return res;
}
```

5.9 nCr mod p in O(1) [SK]

```
// array to store inverse of 1 to N
ll factorialNumInverse[N + 1];

// array to precompute inverse of 1! to N!
ll naturalNumInverse[N + 1];

// array to store factorial of first N numbers
ll fact[N + 1];

// Function to precompute inverse of numbers
void InverseofNumber(ll p)
{
    naturalNumInverse[0] = naturalNumInverse[1] = 1;
    for (int i = 2; i <= N; i++)
        naturalNumInverse[i] = naturalNumInverse[p % i] * (p
            - p / i) % p;
}
// Function to precompute inverse of factorials
void InverseofFactorial(ll p)
{
    factorialNumInverse[0] = factorialNumInverse[1] = 1;

    // precompute inverse of natural numbers
    for (int i = 2; i <= N; i++)
        factorialNumInverse[i] = (naturalNumInverse[i] *
            factorialNumInverse[i - 1]) % p;
}

// Function to calculate factorial of 1 to N
void factorial(ll p)
{
    fact[0] = 1;

    // precompute factorials
    for (int i = 1; i <= N; i++) {
        fact[i] = (fact[i - 1] * i) % p;
    }
}

// Function to return nCr % p in O(1) time
ll Binomial(ll N, ll R, ll p)
{
    // n C r = n!*inverse(r!)*inverse((n-r)!)
    ll ans = ((fact[N] * factorialNumInverse[R])
             % p * factorialNumInverse[N - R])
            % p;
    return ans;
}
```

5.10 Prime Phi Sieve [MB]

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;

struct PrimePhiSieve
{
private:
 ll n;
 vector<ll> primes, phi;
 vector<bool> is_prime;

public:
 PrimePhiSieve() {}

 PrimePhiSieve(ll n)
 {
  this->n = n, is_prime.resize(n + 5, true), phi.resize(n +
      5, 1);
  phi_sieve();
 }
 void phi_sieve()
 {
  is_prime[0] = is_prime[1] = false;

  for (ll i = 1; i <= n; i++)
   phi[i] = i;

  for (ll i = 1; i <= n; i++)
   if (is_prime[i])
   {
    primes.push_back(i);
    phi[i] *= (i - 1), phi[i] /= i;
    for (ll j = i + i; j <= n; j += i)
     is_prime[j] = false, phi[j] /= i, phi[j] *= (i - 1);
   }
}

ll get_divisors_count(int number, int divisor)
{
 return phi[number / divisor];
}

vector<pll> factorize(ll num)
{
 vector<pll> a;
 for (int i = 0; i < (int)primes.size() && primes[i] * 1LL
      * primes[i] <= num; i++)
  if (num % primes[i] == 0)
  {
   int cnt = 0;
   while (num % primes[i] == 0)
    cnt++, num /= primes[i];
   a.push_back({primes[i], cnt});
  }

 if (num != 1)
  a.push_back({num, 1});
 return a;
}

ll get_phi(int n)
{
 return phi[n];
}
// (n/p) * (p-1) => n- (n/p);
void segmented_phi_sieve(ll l, ll r)
{
 vector<ll> current_phi(r - l + 1);
 vector<ll> left_over_prime(r - l + 1);

 for (ll i = l; i <= r; i++)
  current_phi[i - l] = i, left_over_prime[i - l] = i;

 for (ll p : primes)
 {
  ll to = ((l + p - 1) / p) * p;

  if (to == p)
   to += p;

  for (ll i = to; i <= r; i += p)
```

```
  {
   while (left_over_prime[i - l] % p == 0)
    left_over_prime[i - l] /= p;
   current_phi[i - l] -= current_phi[i - l] / p;
  }
 }

 for (ll i = l; i <= r; i++)
 {
  if (left_over_prime[i - l] > 1)
   current_phi[i - l] -= current_phi[i - l] /
        left_over_prime[i - l];
  cout << current_phi[i - l] << endl;
 }
}

ll phi_sqrt(ll n)
{
 ll res = n;

 for (ll i = 1; i * i <= n; i++)
 {
  if (n % i == 0)
  {
   res /= i;
   res *= (i - 1);

   while (n % i == 0)
    n /= i;
  }
 }

 if (n > 1)
  res /= n, res *= (n - 1);
 return res;
}
};
```

5.11 Prime Sieve [MB]

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;

struct PrimeSieve
{
public:
 vector<int> primes;
 vector<bool> isprime;
 int n;

 PrimeSieve() {}

 PrimeSieve(int _n)
 {
  this->n = _n, isprime.resize(_n + 5, true), primes.clear()
       ;
  sieve();
 }

 void sieve()
 {
  isprime[0] = isprime[1] = false;

  primes.push_back(2);
  for (int i = 4; i <= n; i += 2)
   isprime[i] = false;

  for (int i = 3; 1LL * i * i <= n; i += 2)
   if (isprime[i])
    for (int j = i * i; j <= n; j += 2 * i)
     isprime[j] = false;

  for (int i = 3; i <= n; i += 2)
   if (isprime[i])
    primes.push_back(i);
 }

 vector<pll> factorize(ll num)
 {
  vector<pll> a;
  for (int i = 0; i < (int)primes.size() && primes[i] * 1LL
       * primes[i] <= num; i++)
   if (num % primes[i] == 0)
   {
    int cnt = 0;
    while (num % primes[i] == 0)
     cnt++, num /= primes[i];
    a.push_back({primes[i], cnt});
   }

  if (num != 1)
   a.push_back({num, 1});
  return a;
 }

 vector<ll> segemented_sieve(ll l, ll r)
 {
  vector<ll> seg_primes;
  vector<bool> current_primes(r - l + 1, true);
  for (ll p : primes)
  {
   ll to = (l / p) * p;
   if (to < l)
    to += p;
   if (to == p)
    to += p;
   for (ll i = to; i <= r; i += p)
   {
    current_primes[i - l] = false;
   }
  }

  for (ll i = l; i <= r; i++)
  {
   if (i < 2)
    continue;
   if (current_primes[i - l])
   {

    seg_primes.push_back(i);
   }
  }
  return seg_primes;
 }
};
```

5.12 Segmented sieve phi [NK[

```
vector<int64_t> phi_seg;

void seg_sieve_phi(const int64_t a, const int64_t b) {
    phi_seg.assign(b - a + 2, 0);
    vector<int64_t> factor(b - a + 2, 0);
    for (int64_t i = a; i <= b; i++) {
        auto m = i - a + 1;
        phi_seg[m] = i;
        factor[m] = i;
    }
    auto lim = sqrt(b) + 1;
    sieve(lim);
    for (auto p : primes) {
        int64_t a1 = p * ((a + p - 1) / p);
        for (int64_t j = a1; j <= b; j += p) {
```

```
            auto m = j - a + 1;
            while (factor[m] % p == 0) {
                factor[m] /= p;
            }
            phi_seg[m] -= (phi_seg[m] / p);
        }
    }
    for (int64_t i = a; i <= b; i++) {
        auto m = i - a + 1;
        if (factor[m] > 1) {
            phi_seg[m] -= (phi_seg[m] / factor[m]);
            factor[m] = 1;
        }
    }
}
```

5.13 Segmented sieve primes [NK[

```
vector<bool> isprime_seg;
vector<int64_t> seg_primes;

void seg_sieve(const int64_t a, const int64_t b) {
    isprime_seg.assign(b - a + 1, true);
    int lim = sqrt(b) + 1;
    sieve(lim);
    for (auto p : primes) {
        auto a1 = p * max((int64_t)(p), ((a + p - 1) / p));
        for (auto j = a1; j <= b; j += p) {
            isprime_seg[j - a] = false;
        }
    }
    for (auto i = a; i <= b; i++) {
        if (isprime_seg[i - a]) {
            seg_primes.push_back(i);
        }
    }
}
```

5.14 Sieve phi [NK[

```
vector<int> phi;

void sieve_phi(int n) {
    phi.assign(n + 1, 0);
    iota(phi.begin(), phi.end(), 0);
    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i) {
                phi[j] -= (phi[j] / i);
            }
        }
    }
}
```

6 String

6.1 Hashing [MB]

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

const int PRIMES[] = {2147462393, 2147462419, 2147462587,
    2147462633, 2147462747, 2147463167, 2147463203,
    2147463569, 2147463727, 2147463863, 2147464211,
    2147464549, 2147464751, 2147465153, 2147465563,
    2147465599, 2147465743, 2147465953, 2147466457,
    2147466463, 2147466521, 2147466721, 2147467009,
    2147467057, 2147467067, 2147467261, 2147467379,
    2147467463, 2147467669, 2147467747, 2147468003,
    2147468317, 2147468591, 2147468651, 2147468779,
    2147468801, 2147469017, 2147469041, 2147469173,
    2147469229, 2147469593, 2147469881, 2147469983,
    2147470027, 2147470081, 2147470177, 2147470673,
    2147470823, 2147471057, 2147471327, 2147471581,
    2147472137, 2147472161, 2147472689, 2147472697,
    2147472863, 2147473151, 2147473369, 2147473733,
    2147473891, 2147473963, 2147474279, 2147474921,
    2147474929, 2147475107, 2147475221, 2147475347,
    2147475397, 2147475971, 2147476739, 2147476769,
    2147476789, 2147476927, 2147477063, 2147477107,
    2147477249, 2147477807, 2147477933, 2147478017,
    2147478521};

// ll base_pow,base_pow_1;
ll base1 = 43, base2 = 47, mod1 = 1e9 + 7, mod2 = 1e9 + 9;

// **** Enable this function for codeforces
void generateRandomBM()
{
 unsigned int seed = chrono::system_clock::now().
     time_since_epoch().count();
 srand(seed); /// to avoid getting hacked in CF, comment
     this line for easier debugging

 int q_len = (sizeof(PRIMES) / sizeof(PRIMES[0])) / 4;
 base1 = PRIMES[rand() % q_len];
 mod1 = PRIMES[rand() % q_len + q_len];
 base2 = PRIMES[rand() % q_len + 2 * q_len];
 mod2 = PRIMES[rand() % q_len + 3 * q_len];
}

struct Hash
{
public:
 vector<int> base_pow, f_hash, r_hash;
 ll base, mod;

 Hash() {}
 // Update it make it more dynamic like segTree class and
     DSU
 Hash(int mxSize, ll base, ll mod) // Max size
 {
  this->base = base;
  this->mod = mod;
  base_pow.resize(mxSize + 2, 1), f_hash.resize(mxSize + 2,
      0), r_hash.resize(mxSize + 2, 0);

  for (int i = 1; i <= mxSize; i++)
  {
   base_pow[i] = base_pow[i - 1] * base % mod;
  }
 }

 void init(string s)
 {
  int n = s.size();

  for (int i = 1; i <= n; i++)
  {
   f_hash[i] = (f_hash[i - 1] * base + int(s[i - 1])) % mod;
  }

  for (int i = n; i >= 1; i--)
  {
   r_hash[i] = (r_hash[i + 1] * base + int(s[i - 1])) % mod;
  }
 }

 int forward_hash(int l, int r)
 {
```

```
 int h = f_hash[r + 1] - (1LL * base_pow[r - l + 1] *
     f_hash[l]) % mod;
 return h < 0 ? mod + h : h;
}

int reverse_hash(int l, int r)
{
 int h = r_hash[l + 1] - (1LL * base_pow[r - l + 1] *
     r_hash[r + 2]) % mod;
 return h < 0 ? mod + h : h;
}
};

class DHash
{
public:
 Hash sh1, sh2;
 DHash() {}

 DHash(int mx_size)
 {
  sh1 = Hash(mx_size, base1, mod1);
  sh2 = Hash(mx_size, base2, mod2);
 }

 void init(string s)
 {
  sh1.init(s);
  sh2.init(s);
 }

 ll forward_hash(int l, int r)
 {
  return (ll(sh1.forward_hash(l, r)) << 32) | (sh2.
      forward_hash(l, r));
 }

 ll reverse_hash(int l, int r)
 {
  return ((ll(sh1.reverse_hash(l, r)) << 32) | (sh2.
      reverse_hash(l, r)));
 }
};
```

6.2 Hashing [NK]

```
// Primes suitable for use as the constant base in a
    polynomial rolling hash function.
constexpr std::array<int, 10>
    prime_bases = {257, 263, 269, 271, 277, 281, 283, 293,
        307, 311};
// Primes suitable for use as modulus.
constexpr std::array<int, 10>
    prime_moduli = {1000000007, 1000000009, 1000000021,
        1000000033, 1000000087,
                    1000000093, 1000000097, 1000000103,
                        1000000123, 1000000181};

/**
 * @brief A data structure for computing polynomial hashes
     of sequence keys.
 * For a given key defined as an integral sequence of n
     elements S[0], S[1], ...,
 * S[n - 1], this structure builds and stores for each
     prefix S[0...i] the hash value
 * H(i) = S[0] * B^i + S[1] * B^(i - 1) + ... + S[i] * B^0,
     modulo M.
 * @tparam Base The base B. Should be a prime to reduce
     chances of collision.
 * @tparam Modulus The modulus M. Should be a prime to
     reduce chances of collision.
 */
template <std::uint64_t Base, std::uint64_t Modulus>
class Polynomial_hasher {
public:
    using int_type = std::uint64_t;
    using value_type = int_type;
    using size_type = std::size_t;

    static constexpr int_type B = Base;
    static constexpr int_type M = Modulus;

protected:
    // Base power
    static std::vector<int_type> bpow_;
    // Prefix hash
    std::vector<int_type> pref_hash_;
    // Suffix hash
    std::vector<int_type> suff_hash_;
    // Flag for hashing bidirectionally
    bool bidir_ = false;

public:
    /**
     * @brief Default constructor
     */
    Polynomial_hasher() {}

    /**
     * @brief Constructors and builds the hash from a range (
         a "key").
     * @tparam InputIter Type of the iterator of the range
     * @param from Iterator pointing to the start of the
         range
     * @param until Iterator pointing to the end (one past
         the last element) of the range
     * @param bidir Flag for hashing bidirectionally
     */
    template <class InputIter>
    Polynomial_hasher(InputIter from, InputIter until, bool
         bidir = false) {
        build_hash(from, until, bidir);
    }

    /**
     * @brief Builds the hash from a range (a "key").
     * @tparam InputIter Type of the iterator of the range
     * @param from Iterator pointing to the start of the
         range
     * @param until Iterator pointing to the end (one past
         the last element) of
     * the range
     * @param bidir Flag for hashing bidirectionally
     */
    template <class InputIter>
    void build_hash(InputIter from, InputIter until, bool
         bidir = false) {
        const auto n = std::distance(from, until);
        while (bpow_.size() < n) {
            bpow_.push_back((bpow_.back() * B) % M);
        }
        // Build forward hash
        {
            pref_hash_.resize(n + 1);
            pref_hash_[0] = 0;
            auto it = from;
            for (size_type i = 0; i < n; ++i) {
                pref_hash_[i + 1] =
                    (((pref_hash_[i] * B) % M) + static_cast<
                        int_type>(*it)) % M;
                ++it;
            }
        }
        // Set and test flag, and build reverse hash
        bidir_ = bidir;
        if (bidir_) {
            suff_hash_.resize(n + 1);
            suff_hash_[n] = 0;
            auto it = prev(until);
```

```
        for (size_type i = n; i; --i) {
            suff_hash_[i - 1] =
                (((suff_hash_[i] * B) % M) + static_cast<
                    int_type>(*it)) % M;
            --it;
        }
    }
}

/**
 * @brief Returns the polynomial hash value of the
     subsegment S[i], S[i + 1], ...,
 * S[i + n - 1], which is the value S[i] * B^(n - 1) + S[
     i + 1] * B^(n - 2) +
 * ... + S[i + n - 1] * B^0, modulo M.
 * @param i Starting index/position of the subsegment
 * @param n Length of the subsegment
 */
value_type get(size_type i = 0,
               size_type n = std::numeric_limits<size_type
                   >::max()) const {
    assert(i < pref_hash_.size());
    n = std::min(n, pref_hash_.size() - 1 - i);
    return (pref_hash_[i + n] - ((pref_hash_[i] * bpow_[n
        ]) % M) + M) % M;
}

/**
 * @brief Returns the polynomial hash value of the
     subsegment S[i], S[i + 1], ...,
 * S[i + n - 1] in reverse order, which is the value S[i]
     * B^i + S[i + 1] *
 * B^(i + 1) + ... + S[i + n - 1] * B^(i + n - 1), modulo
     M.
 * @param i Starting index/position of the subsegment
 * @param n Length of the subsegment
 */
value_type get_rev(size_type i = 0,
                   size_type n = std::numeric_limits<
                       size_type>::max()) const {
    assert(bidir_);
    assert(i < suff_hash_.size());
    n = std::min(n, suff_hash_.size() - 1 - i);
    return (suff_hash_[i] - ((suff_hash_[i + n] * bpow_[n
        ]) % M) + M) % M;
}

/**
 * @brief Erases hash values of all prefixes (and
     suffixes if hashed
     * bidirectionally) calling `clear()` on the internal
         vector(s). Resets
     * bidirectional flag.
     */
    void clear() {
        pref_hash_.clear();
        suff_hash_.clear();
        bidir_ = false;
    }

    /**
     * @brief Number of elements in the hashed key.
     */
    size_type size() const { return pref_hash_.size() ?
        pref_hash_.size() - 1 : 0; }

    /**
     * @brief Returns true if no hash values are stored.
     */
    bool empty() const { return pref_hash_.empty(); }

    /**
     * @brief Returns true if the stored hash value is
         bidirectional (i.e., both
     * `hash` and `hash_rev` can be called).
     */
    bool bidirectional() const { return bidir_; }
};

template <std::uint64_t Base, std::uint64_t Modulus>
std::vector<std::uint64_t> Polynomial_hasher<Base, Modulus
    >::bpow_ = {1ULL};

using Hasher0 = Polynomial_hasher<prime_bases[0],
    prime_moduli[0]>;
using Hasher1 = Polynomial_hasher<prime_bases[1],
    prime_moduli[1]>;
```

6.3 String Hashing With Point Updates [SA]

```
struct Node {
    int64_t fwd, rev;
    int len;
    Node(int64_t f, int64_t r, int l) {
        fwd = f, rev = r, len = l;
    }
    Node() {
        fwd = rev = len = 0;
    }
};

const int BASE = 47, MX_N = 1E5 + 5, M = 1E9 + 7;
string a;
Node st[4 * MX_N];
int64_t expo[MX_N];// TODO: compute this beforehand

void build(int node, int tL, int tR) {
    if (tL == tR) {
        st[node] = Node(a[tL], a[tL], 1);
        return;
    }
    int mid = (tL + tR) / 2;
    int left = 2 * node, right = 2 * node + 1;
    build(left, tL, mid);
    build(right, mid + 1, tR);
    st[node] = Node((st[left].fwd * expo[st[right].len] + st[
        right].fwd) % M,
                    (st[right].rev * expo[st[left].len] + st[
                        left].rev) % M,
                    st[left].len + st[right].len);
}

void update(int node, int tL, int tR, int i, int64_t v) {
    if (tL >= i && tR <= i) {
        st[node] = Node(v, v, 1);
        return;
    }
    if (tR < i || tL > i) return;

    int mid = (tL + tR) / 2;
    int left = 2 * node, right = 2 * node + 1;
    update(left, tL, mid, i, v);
    update(right, mid + 1, tR, i, v);
    st[node] = Node((st[left].fwd * expo[st[right].len] + st[
        right].fwd) % M,
                    (st[right].rev * expo[st[left].len] + st[
                        left].rev) % M,
                    st[left].len + st[right].len);
}

Node query(int node, int tL, int tR, int qL, int qR) {
    if (tL >= qL && tR <= qR) {
        return Node(st[node].fwd, st[node].rev, st[node].len)
            ;
    }
    if (tR < qL || tL > qR) {
        return Node(0, 0, 0);
```

```
    }
    int mid = (tL + tR) / 2;
    auto QL = query(2 * node, tL, mid, qL, qR);
    auto QR = query(2 * node + 1, mid + 1, tR, qL, qR);
    return Node((QL.fwd * expo[QR.len] + QR.fwd) % M, (QR.rev
          * expo[QL.len] + QL.rev) % M, QL.len + QR.len);
}
```

6.4 Z-Function [MB]

```
#include<bits/stdc++.h>

/*
 tested by ac
 submission: https://codeforces.com/contest/432/submission
      /145953901
 problem: https://codeforces.com/contest/432/problem/D
*/
std::vector<int> z_function(const std::string &s)
{
 int n = (int)s.size();
 std::vector<int> z(n, 0);
 for (int i = 1, l = 0, r = 0; i < n; i++)
 {
  if (i <= r)
   z[i] = std::min(r - i + 1, z[i - l]);
  while (i + z[i] < n && s[z[i]] == s[i + z[i]])
   z[i]++;
  if (i + z[i] - 1 > r)
   l = i, r = i + z[i] - 1;
 }
 return z;
}
```