Team Notebook

June 24, 2022

Contents

1 Algebra

1.1 Fundamentals

1.1.1 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;
}
```

1.1.2 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;
}
```

1.2 Modular Arithmetic

1.2.1 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;
}
```

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 Concepts

3.1 General

3.1.1 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;
    }
};
```

3.1.2 UnorderedMap[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;
```

3.2 Graph

3.2.1 edgeRemoveCC[MB]

```cpp
#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 Successfully 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;
}
```

3.2.2 treerooting[MB]

```cpp
#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;
}
```

4 Data Structures

4.1 Graph

4.1.1 DSU[MB]

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

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

public:
 DSU() {}

 //0 based
 DSU(int mx_size)
 {
  //Default empty
  p.resize(mx_size, 0), csz.resize(mx_size, 0);

  init(mx_size);
 }

 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;
 }
};
```

4.1.2 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 = 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(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)];
 }
};
```

4.2 NumberTheory

4.2.1 Combinatrics[MB]

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

using namespace std;

typedef long long ll;

const int N = 1e6, MOD = 998244353;

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;
 }
};
```

4.2.2 ModInt[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>;
```

4.2.3 PrimePhiSieve[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;
 }
};
```

4.2.4 PrimeSieve[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 (int i = l; i <= r; i++)
  {
   if (i < 2)
    continue;
   if (current_primes[i - l])
   {

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

4.3 QueryUpdate

4.3.1 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;
  }
 }
};
```

4.3.2 LazySegTree[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)
  {
   T 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)]);
 }

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();
  for (int i = 0; i < n; i++)
   set(i, i, arr[i]);
 }

 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;
}
```

4.3.3 MosAlgo[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;
}
```

4.3.4 SegTree[MB]

4.3.5 SparseTable[MB]

4.4 String

4.4.1 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)));
 }
};
```

4.5 Trees

4.5.1 Disjoint Set Union (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;
    }
};
```

5 Starter

5.1 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>;
```

5.2 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...);
}
```

5.3 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;
}
```

5.4 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;
}
```

6 String Processing

6.1 Fundamentals

6.1.1 Polynomial Rolling Hash (String Hashing) [NK]

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

namespace hash_utils {
    constexpr std::array<int, 10U>
        bases = {257, 263, 269, 271, 277, 281, 283, 293, 307,
              311};
    constexpr std::array<int, 10U>
        moduli = {1000000007, 1000000009, 1000000021,
             1000000033, 1000000087,
                  1000000093, 1000000097, 1000000103,
                       1000000123, 1000000181};
} // namespace hash_utils



template <std::size_t Dim, class Tp>
using require_valid_hash_params_t =
    std::enable_if_t<(1U <= Dim && Dim <= 3U) &&
                     (std::is_same<Tp, int>::value ||
                      std::is_same<Tp, std::int64_t>::value)>;

template <std::size_t Dim, class Tp = int,
          class = require_valid_hash_params_t<Dim, Tp>>
class Rolling_hash {
private:
using Vec = std::vector<Tp>;

static std::conditional_t<Dim == 1U, Tp, std::array<Tp,
     Dim>> base_;
static std::conditional_t<Dim == 1U, Tp, std::array<Tp,
     Dim>> mod_;
static std::conditional_t<Dim == 1U, Vec, std::array<Vec,
      Dim>> pow_;
static std::conditional_t<Dim == 1U, Vec, std::array<Vec,
      Dim>> inv_;
static int nchanges_;
static bool ischanged_;

std::conditional_t<Dim == 1U, Vec, std::array<Vec, Dim>>
     pref_;
std::conditional_t<Dim == 1U, Vec, std::array<Vec, Dim>>
     suff_;
int changeid_;
bool ishashed_;
bool isbidirect_;

template <class T = Tp,
          std::enable_if_t<std::is_same<T, int>::value>* =
                nullptr>
static constexpr Tp mul(const Tp& a, const Tp& b, const
     Tp& mod) {
    return ((static_cast<std::int64_t>(a) * b) % mod);
}

template <class T = Tp,
          std::enable_if_t<std::is_same<T, std::int64_t>::
               value>* = nullptr>
static constexpr Tp mul(const Tp& a, const Tp& b, const
     Tp& mod) {
    long double prod = static_cast<long double>(a) * b;
    std::int64_t quot = prod / mod;
    return (prod - (quot * mod) + 1e-6);
}

template <class T = Tp,
          std::enable_if_t<std::is_same<T, int>::value>* =
                nullptr>
static constexpr Tp add(const Tp& a, const Tp& b, const
     Tp& mod) {
    return ((static_cast<std::int64_t>(a) + b) % mod);
}

template <class T = Tp,
          std::enable_if_t<std::is_same<T, std::int64_t>::
               value>* = nullptr>
    static constexpr Tp add(const Tp& a, const Tp& b, const
         Tp& mod) {
        long double sum = static_cast<long double>(a) + b;
        std::int64_t quot = sum / mod;
        return (sum - (quot * mod) + 1e-6);
    }

    static constexpr Tp inverse(Tp a, const Tp& mod) {
        Tp b = mod, x = 1, y = 0;
        Tp x1 = 0, y1 = 1, tmp = 0, q = 0;
        while (b > 0) {
            q = a / b;
            tmp = a, a = b, b = tmp - (q * b);
            tmp = x, x = x1, x1 = tmp - (q * x1);
            tmp = y, y = y1, y1 = tmp - (q * y1);
        }
        assert(a == 1);
        if (x < 0) x += mod;
        return x;
    }

    template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
         >* = nullptr>
    static constexpr void normalize_kernel(std::size_t len) {
        if (!(ischanged_ || pow_.size() < len)) {
            return;
        }
        auto cur_len = ischanged_ ? 0U : pow_.size();
        pow_.resize(len);
        pow_[0U] = 1;
        for (auto i = ((cur_len == 0U) ? 1U : cur_len); i <
             len; ++i) {
            pow_[i] = mul(pow_[i - 1U], base_, mod_);
        }
        inv_.resize(len);
        for (auto i = cur_len; i < len; ++i) {
            inv_[i] = inverse(pow_[i], mod_);
        }
        ischanged_ = false;
    }

    template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
         >* = nullptr>
    static constexpr void normalize_kernel(std::size_t len) {
        if (!(ischanged_ || pow_[0U].size() < len)) {
            return;
        }
        auto cur_len = ischanged_ ? 0U : pow_[0U].size();
        auto start_idx = (cur_len == 0U) ? 1U : cur_len;
        for (auto k = 0U; k < Dim; ++k) {
            const auto& m = mod_[k];
            pow_[k].resize(len);
            pow_[k][0U] = 1;
            for (auto i = start_idx; i < len; ++i) {
                pow_[k][i] = mul(pow_[k][i - 1U], base_[k], m)
                     ;
            }
            inv_[k].resize(len);
            for (auto i = cur_len; i < len; ++i) {
                inv_[k][i] = inverse(pow_[k][i], m);
            }
        }
        ischanged_ = false;
    }

public:
    Rolling_hash() : ishashed_(false), isbidirect_(false),
         changeid_(-1) {}

    template <class InputIter,
             std::_RequireInputIter<InputIter>* = nullptr>
    Rolling_hash(InputIter first, InputIter last, bool
         bidirectional = false)
        : Rolling_hash() { hash(first, last, bidirectional);
            }

    template <class InputIter,
             std::_RequireInputIter<InputIter>* = nullptr,
             std::size_t KK = Dim, std::enable_if_t<KK == 1U
                  >* = nullptr>
    void hash(InputIter first, InputIter last,
             bool bidirectional = false) {
        const std::size_t len = std::distance(first, last);
        assert(len > 0U);
        normalize_kernel(len);

        isbidirect_ = bidirectional;
        changeid_ = nchanges_;

        auto i = 0U, j = 0U;

        pref_.resize(len);
        pref_[0U] = static_cast<Tp>(*first) % mod_;
        i = 1U;
        for (auto it = next(first); it != last; ++it) {
            pref_[i] = add(pref_[i - 1U], mul(static_cast<Tp
                 >(*it), pow_[i], mod_), mod_);
            ++i;
        }
        if (!bidirectional) {
            ishashed_ = true;
            return;
        }

        suff_.resize(len);
        const auto &prev_first = prev(first), prev_last =
             prev(last);
        suff_[len - 1U] = static_cast<Tp>(*prev_last) % mod_;
        i = len - 2U, j = 1U;
        for (auto it = prev(prev_last); it != prev_first; --
             it) {
            suff_[i] = add(suff_[i + 1U], mul(static_cast<Tp
                 >(*it), pow_[j], mod_), mod_);
            --i, ++j;
        }

        ishashed_ = true;
    }

    template <class InputIter,
             std::_RequireInputIter<InputIter>* = nullptr,
             std::size_t KK = Dim, std::enable_if_t<KK != 1U
                  >* = nullptr>
    void hash(InputIter first, InputIter last,
             bool bidirectional = false) {
        const std::size_t len = std::distance(first, last);
        assert(len > 0U);
        normalize_kernel(len);

        isbidirect_ = bidirectional;
        changeid_ = nchanges_;

        auto i = 0U, j = 0U;
        const auto &prev_first = prev(first), prev_last =
             prev(last);

        for (auto k = 0U; k < Dim; ++k) {
            const auto& m = mod_[k];
            pref_[k].resize(len);
            pref_[k][0U] = static_cast<Tp>(*first) % m;
            i = 1U;
            for (auto it = next(first); it != last; ++it) {
                pref_[k][i] = add(pref_[k][i - 1U],
                                mul(static_cast<Tp>(*it), pow_
                                     [k][i], m), m);
                ++i;
            }
            if (!bidirectional) {
                continue;
        }
        suff_[k].resize(len);
        suff_[k][len - 1U] = static_cast<Tp>(*prev_last)
             % m;
        i = len - 2U, j = 1U;
        for (auto it = prev(prev_last); it != prev_first;
              --it) {
            suff_[k][i] = add(suff_[k][i + 1U],
                              mul(static_cast<Tp>(*it), pow_
                                    [k][j], m), m);
            --i, ++j;
        }
    }

    ishashed_ = true;
}

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
     >* = nullptr>
Tp get(std::size_t pos = 0U, std::size_t len = SIZE_MAX)
     const {
    if (!ishashed_) {
        throw std::runtime_error("Not hashed yet");
    }
    if (changeid_ != nchanges_) {
        throw std::runtime_error("At least one of ‘base‘
             and ‘modulus‘ has \
        changed and hence this instance can no longer be
             sliced");
    }
    if (pos >= pref_.size()) {
        throw std::out_of_range("Starting index is out of
              range");
    }
    if (len == 0U) {
        return 0;
    }

    std::size_t r = std::min(pos + len, pref_.size()) - 1
         U;
    if (pos == 0U) {
        return pref_[r];
    }
    return mul((pref_[r] - pref_[pos - 1U] + mod_) % mod_
         , inv_[pos], mod_);
}

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
     >* = nullptr>
```

```
std::array<Tp, Dim> get(std::size_t pos = 0U, std::size_t
      len = SIZE_MAX) const {
    if (!ishashed_) {
        throw std::runtime_error("Not hashed yet");
    }
    if (changeid_ != nchanges_) {
        throw std::runtime_error("At least one of ‘base‘
             and ‘modulus‘ has \
        changed and hence this instance can no longer be
             sliced");
    }
    if (pos >= pref_[0U].size()) {
        throw std::out_of_range("Starting index is out of
              range");
    }
    if (len == 0U) {
        return std::array<Tp, Dim>({});
    }

    std::size_t r = std::min(pos + len, pref_[0U].size())
          - 1U;
    std::array<Tp, Dim> res;
    if (pos == 0U) {
        for (auto k = 0U; k < Dim; ++k) {
            res[k] = pref_[k][r];
        }
        return res;
    }
    for (auto k = 0U; k < Dim; ++k) {
        const auto& m = mod_[k];
        res[k] = mul((pref_[k][r] - pref_[k][pos - 1U] +
             m) % m,
                     inv_[k][pos], m);
    }
    return res;
}

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
     >* = nullptr>
Tp getrev(std::size_t pos = 0U, std::size_t len =
     SIZE_MAX) const {
    if (!ishashed_) {
        throw std::runtime_error("Not hashed yet");
    }
    if (!isbidirect_) {
        throw std::runtime_error("Not hashed
             bidirectionally");
    }
    if (changeid_ != nchanges_) {
```

```
        throw std::runtime_error("At least one of ‘base‘
             and ‘modulus‘ has \
        changed and hence this instance can no longer be
             sliced");
    }
    if (pos >= suff_.size()) {
        throw std::out_of_range("Starting index is out of
              range");
    }
    if (len == 0U) {
        return 0;
    }

    std::size_t r = std::min(pos + len, suff_.size()) - 1
         U;
    auto rem = suff_.size() - 1U - r;
    if (rem == 0U) {
        return suff_[pos];
    }
    return mul((suff_[pos] - suff_[r + 1U] + mod_) % mod_
         , inv_[rem], mod_);
}

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
     >* = nullptr>
std::array<Tp, Dim> getrev(std::size_t pos = 0U, std::
     size_t len = SIZE_MAX) const {
    if (!ishashed_) {
        throw std::runtime_error("Not hashed yet");
    }
    if (!isbidirect_) {
        throw std::runtime_error("Not hashed
             bidirectionally");
    }
    if (changeid_ != nchanges_) {
        throw std::runtime_error("At least one of ‘base‘
             and ‘modulus‘ has \
        changed and hence this instance can no longer be
             sliced");
    }
    if (pos >= suff_[0U].size()) {
        throw std::out_of_range("Starting index is out of
              range");
    }
    if (len == 0U) {
        return std::array<Tp, Dim>({});
    }

    std::size_t r = std::min(pos + len, suff_[0U].size())
          - 1U;
    auto rem = suff_[0U].size() - 1U - r;
    std::array<Tp, Dim> res;
    if (rem == 0U) {
        for (auto k = 0U; k < Dim; ++k) {
            res[k] = suff_[k][pos];
        }
        return res;
    }
    for (auto k = 0U; k < Dim; ++k) {
        const auto& m = mod_[k];
        res[k] = mul((suff_[k][pos] - suff_[k][r + 1U] +
             m) % m,
                   inv_[k][rem], m);
    }
    return res;
}

bool is_hashed() const { return ishashed_; }
bool is_bidirectional() const { return isbidirect_; }

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
     >* = nullptr>
std::size_t size() const { return pref_.size(); }

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
     >* = nullptr>
std::size_t size() const { return pref_[0U].size(); }

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
     >* = nullptr>
static constexpr Tp base() { return base_; }

template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
     >* = nullptr>
static constexpr Tp base(std::size_t i) { return base_[i
     ]; }

template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
     >* = nullptr>
static constexpr void base(Tp new_base) {
    if (new_base <= 1) {
        throw std::invalid_argument("‘new_base‘ must be
             greater than 1");
    }
    base_ = new_base;
    ischanged_ = true;
    ++nchanges_;
}
```

```
    template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
         >* = nullptr>
    static constexpr void base(std::size_t i, Tp new_base) {
        if (new_base <= 1) {
            throw std::invalid_argument("‘new_base‘ must be
                 greater than 1");
        }
        base_[i] = new_base;
        ischanged_ = true;
        ++nchanges_;
    }

    template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
         >* = nullptr>
    static constexpr Tp modulus() { return mod_; }

    template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
         >* = nullptr>
    static constexpr Tp modulus(std::size_t i) {
        return mod_[i];
    }

    template <std::size_t KK = Dim, std::enable_if_t<KK == 1U
         >* = nullptr>
    static constexpr void modulus(Tp new_modulus) {
        if (new_modulus <= 1) {
            throw std::invalid_argument("‘new_modulus‘ must
                 be greater than 1");
        }
        mod_ = new_modulus;
        ischanged_ = true;
        ++nchanges_;
    }

    template <std::size_t KK = Dim, std::enable_if_t<KK != 1U
         >* = nullptr>
    static constexpr void modulus(std::size_t i, Tp
         new_modulus) {
        if (new_modulus <= 1) {
            throw std::invalid_argument("‘new_modulus‘ must
                 be greater than 1");
        }
        mod_[i] = new_modulus;
        ischanged_ = true;
        ++nchanges_;
    }
};

template <std::size_t Dim, class Tp, class Enabler>
int Rolling_hash<Dim, Tp, Enabler>::nchanges_ = 0;
```

```
template <std::size_t Dim, class Tp, class Enabler>
bool Rolling_hash<Dim, Tp, Enabler>::ischanged_ = false;

#ifndef ROLLING_HASH_PARTIAL_SPEC_INIT
#define ROLLING_HASH_PARTIAL_SPEC_INIT(Tp)
                                \
    template <>

         \
    std::array<Tp, 3U> Rolling_hash<3U, Tp>::base_ = {257,
         263, 269};  \
    template <>

         \
    std::array<Tp, 3U> Rolling_hash<3U, Tp>::mod_ =
         {1000000007, 1000000009, \
                                                1000000021};


                                                     \
    template <>

         \
    std::array<std::vector<Tp>, 3U> Rolling_hash<3U, Tp>::
         pow_ = {};   \
    template <>

         \
    std::array<std::vector<Tp>, 3U> Rolling_hash<3U, Tp>::
         inv_ = {};   \


    template <>

         \
    std::array<Tp, 2U> Rolling_hash<2U, Tp>::base_ = {257,
         263};       \
    template <>

         \
    std::array<Tp, 2U> Rolling_hash<2U, Tp>::mod_ =
         {1000000007, 1000000009}; \
    template <>

         \
    std::array<std::vector<Tp>, 2U> Rolling_hash<2U, Tp>::
         pow_ = {};   \
    template <>

         \
std::array<std::vector<Tp>, 2U> Rolling_hash<2U, Tp>::
     inv_ = {};   \


template <>

     \
Tp Rolling_hash<1U, Tp>::base_ = 257;
                                  \
template <>

     \
Tp Rolling_hash<1U, Tp>::mod_ = 1000000007;
                            \
template <>

     \
std::vector<Tp> Rolling_hash<1U, Tp>::pow_ = {};
                       \
template <>
        \
   std::vector<Tp> Rolling_hash<1U, Tp>::inv_ = {};
        \
ROLLING_HASH_PARTIAL_SPEC_INIT(int)
ROLLING_HASH_PARTIAL_SPEC_INIT(std::int64_t)
#endif

template <class Tp>
using single_hash = Rolling_hash<1U, Tp>;
template <class Tp>
using double_hash = Rolling_hash<2U, Tp>;
template <class Tp>
using triple_hash = Rolling_hash<3U, Tp>;
```

6.2 $\mathbf{z}_{f}unction[MB]$

```
#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;
}
```