Contents

1. Contest

1.1. rng.h

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

int random(int a, int b) {
  if (a > b)
    return 0;
  return a + rng() % (b - a + 1);
}
double random_double(double a, double b) {
  return a + (b - a) * (rng() / (double)rng.max());
}
```

1.2. template.h

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

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, less_equal<int>, rb_tree_tag,
             tree_order_statistics_node_update>
    ordered_multiset;
typedef tree<int, null_type, less<int>, rb_tree_tag,
             tree_order_statistics_node_update>
    ordered_set;
// find_by_order returns iterator to kth largest (0-indexed)
// order_of_key returns number of elements strictly less than
given value -->
// basically index (0-indexed) for multiset, to erase use
upper_bound.
// Upper_bound lower_bound exchange their roles
#define ONLINE_JUDGE
#ifndef ONLINE_JUDGE
#define db(x) cerr << #x << " == " << x << endl
#define dbs(x) cerr << x << endl
#else
#define db(x) ((void)0)
#define dbs(x) ((void)0)
#endif

#define int long long
#define fast()
\
  ios_base::sync_with_stdio(0);
\
  cin.tie(NULL);
\
  cout.tie(NULL);
#define fr(i, a, b) for (int i = (a); i < (int)(b); ++i)
#define pb push_back
#define prDouble(x) cout << fixed << setprecision(10) << x
int M = 1e9 + 7;
#define all(x) x.begin(), x.end()
#define allr(x) x.rbegin(), x.rend()
#define sz(x) (int)x.size()
```

```
void solve() {}
signed main() {
  fast();
  int t = 1;
  cin >> t;
  while (t--) {
    solve();
  }
  return 0;
}
```

2. Data Structures

2.1. LazySegTree.h

```
template <typename T, typename U> struct seg_tree_lazy {
  int S, H;
  T zero;
  vector<T> value;
  U noop;
  vector<bool> dirty;
  vector<U> prop;
  seg_tree_lazy(int _S, T _zero = T(), U _noop = U()) {
    zero = _zero, noop = _noop;
    for (S = 1, H = 1; S < _S;)
      S *= 2, H++;
    value.resize(2 * S, zero);
    dirty.resize(2 * S, false);
    prop.resize(2 * S, noop);
  }
  void set_leaves(vector<T> &leaves) {
    copy(leaves.begin(), leaves.end(), value.begin() + S);
    for (int i = S - 1; i > 0; i--)
      value[i] = value[2 * i] + value[2 * i + 1];
  }
  void apply(int i, U &update) {
    value[i] = update(value[i]);
    if (i < S) {
      prop[i] = prop[i] + update;
      dirty[i] = true;
    }
  }
  void rebuild(int i) {
    for (int l = i / 2; l; l /= 2) {
      T combined = value[2 * l] + value[2 * l + 1];
      value[l] = prop[l](combined);
    }
  }
  void propagate(int i) {
    for (int h = H; h > 0; h--) {
      int l = i >> h;
      if (dirty[l]) {
        apply(2 * l, prop[l]);
        apply(2 * l + 1, prop[l]);
        prop[l] = noop;
        dirty[l] = false;
      }
    }
  }
  void upd(int i, int j, U update) {
    i += S, j += S;
    propagate(i), propagate(j);
    for (int l = i, r = j; l <= r; l /= 2, r /= 2) {
      if ((l & 1) == 1)
        apply(l++, update);
      if ((r & 1) == 0)
        apply(r--, update);
    }
    rebuild(i), rebuild(j);
  }
  T query(int i, int j) {
    i += S, j += S;
    propagate(i), propagate(j);
```

```
    T res_left = zero, res_right = zero;
    for (; i <= j; i /= 2, j /= 2) {
      if ((i & 1) == 1)
        res_left = res_left + value[i++];
      if ((j & 1) == 0)
        res_right = value[j--] + res_right;
    }
    return res_left + res_right;
  }
};
struct node {
  int sum, width;
  node operator+(const node &n) {
    // change 1
    return {sum + n.sum, width + n.width};
  }
};
struct update {
  bool type; // 0 for add, 1 for reset
  int value;
  node operator()(const node &n) {
    // change 2
    if (type)
      return {n.width * value, n.width};
    else
      return {n.sum + n.width * value, n.width};
  }
  update operator+(const update &u) {
    // change 3
    if (u.type)
      return u;
    return {type, value + u.value};
  }
};
// Example Initialization
// seg_tree_lazy<node, update> lst(size, {0, 1}, {0, 0});
// lst.set_leaves(leaves);
// lst.upd(l, r, {0, value});
// auto result = lst.query(l, r);
```

2.2. Merge_sort_tree_pbds.h

```
// Merge Sort Tree with point updates - paste into a .cpp file
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace std;
using namespace __gnu_pbds;

template <typename T>
using ordered_set = tree<pair<T, int>, null_type, less<pair<T,
int>>,
                         rb_tree_tag,
tree_order_statistics_node_update>;

class MergeSortTree {
  int n;
  vector<ordered_set<int>> st;
  vector<int> arr;

  void insert_at(int node, int l, int r, int pos, int val) {
    st[node].insert({val, pos});
    if (l == r)
      return;
    int mid = (l + r) >> 1;
    if (pos <= mid)
      insert_at(node << 1, l, mid, pos, val);
    else
      insert_at(node << 1 | 1, mid + 1, r, pos, val);
  }

  void erase_at(int node, int l, int r, int pos, int val) {
    st[node].erase({val, pos});
```

```
    if (l == r)
      return;
    int mid = (l + r) >> 1;
    if (pos <= mid)
      erase_at(node << 1, l, mid, pos, val);
    else
      erase_at(node << 1 | 1, mid + 1, r, pos, val);
  }

  int query_le_rec(int node, int l, int r, int ql, int qr, int
k) {
    if (qr < l || r < ql)
      return 0;
    if (ql <= l && r <= qr) {
      return (int)st[node].order_of_key({k + 1,
numeric_limits<int>::min()});
    }
    int mid = (l + r) >> 1;
    return query_le_rec(node << 1, l, mid, ql, qr, k) +
           query_le_rec(node << 1 | 1, mid + 1, r, ql, qr, k);
  }

public:
  MergeSortTree() : n(0) {}

  MergeSortTree(const vector<int> &a) { build(a); }

  void build(const vector<int> &a) {
    arr = a;
    n = (int)arr.size();
    if (n == 0)
      return;
    st.assign(4 * n, ordered_set<int>());
    for (int i = 0; i < n; ++i)
      insert_at(1, 0, n - 1, i, arr[i]);
  }

  void point_update(int pos, int newVal) {
    if (pos < 0 || pos >= n)
      return;
    int old = arr[pos];
    if (old == newVal)
      return;
    erase_at(1, 0, n - 1, pos, old);
    insert_at(1, 0, n - 1, pos, newVal);
    arr[pos] = newVal;
  }

  int count_le(int l, int r, int k) {
    if (l < 0)
      l = 0;
    if (r >= n)
      r = n - 1;
    if (l > r || n == 0)
      return 0;
    return query_le_rec(1, 0, n - 1, l, r, k);
  }

  int count_lt(int l, int r, int k) { return count_le(l, r, k -
1); }

  int count_gt(int l, int r, int k) { return (r - l + 1) -
count_le(l, r, k); }

  const vector<int> &data() const { return arr; }
};

// Example usage:
// int main() {
//     vector<int> a = {5,1,7,3,4};
//     MergeSortTree mst(a);
//     cout << mst.count_le(1,4,3) << "\n";
```

```
//     mst.point_update(2,2);
//     cout << mst.count_le(1,4,3) << "\n";
// }
```

2.3. Mos.h

```
int BLOCK = DO_NOT_FORGET_TO_CHANGE_THIS;
struct Query {
  int l, r, id;
  Query(int _l, int _r, int _id) : l(_l), r(_r), id(_id) {}
  bool operator<(Query &o) {
    int mblock = l / BLOCK, oblock = o.l / BLOCK;
    return (mblock < oblock) or
           (mblock == oblock and mblock % 2 == 0 and r < o.r)
or
           (mblock == oblock and mblock % 2 == 1 and r > o.r);
  };
};
void solve() {
  vector<Query> queries;
  queries.reserve(q);
  for (int i = 0; i < q; i++) {
    int l, r;
    cin >> l >> r;
    l--, r--;
    queries.emplace_back(l, r, i);
  }
  sort(all(queries));
  int ans = 0;
  auto add = [&](int v) {};
  auto rem = [&](int v) {};
  vector<int> out(q); // Change out type if necessary
  int cur_l = 0, cur_r = -1;
  for (auto &[l, r, id] : queries) {
    while (cur_l > l)
      add(--cur_l);
    while (cur_l < l)
      rem(cur_l++);
    while (cur_r < r)
      add(++cur_r);
    while (cur_r > r)
      rem(cur_r--);
    out[id] = ans;
  }
}
```

2.4. SegTree.h

```
struct segtree {
  typedef int T;
  // for max segtree, set unit = INT_MIN and f(a,b) = max(a,b)
  static constexpr T unit = 0;    // identity for sum
  T f(T a, T b) { return a + b; } // (any associative fn)
  vector<T> s;
  int n;
  segtree(int n = 0, T def = unit) : s(2 * n, def), n(n) {}
  void update(int pos, T val) {
    for (s[pos += n] = val; pos /= 2;)
      s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
  }
  T query(int b, int e) { // query [b, e)
    T ra = unit, rb = unit;
    for (b += n, e += n; b < e; b /= 2, e /= 2) {
      if (b % 2)
        ra = f(ra, s[b++]);
      if (e % 2)
        rb = f(s[--e], rb);
    }
    return f(ra, rb);
  }
};
```

2.5. fenwick.h

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

/*
  FenwickRange<T>
  - 0-based public indexing for callers
  - Internally uses 1-based Fenwick arrays
  - Methods:
      FenwickRange(int n)              : create size n
(elements indexed 0..n-1)
      void range_add(int l, int r, T v): add v to every element
in [l, r]
  (0-based inclusive) T prefix_sum(int idx)            : sum of
elements
  [0..idx] (0-based). idx < 0 -> 0 T range_sum(int l, int
r)        : sum of
  elements [l..r] (0-based) T point_query(int idx)           :
value at index
  idx (0-based)
*/
template <typename T = long long> class FenwickRange {
public:
  explicit FenwickRange(int n = 0)
      : n_(n), B1(n_ + 1, T(0)), B2(n_ + 1, T(0)) {}

  int size() const noexcept { return n_; }

  // add val to every element in [l, r] (0-based indices)
  void range_add(int l, int r, T val) {
    if (n_ == 0)
      return;
    if (l < 0)
      l = 0;
    if (r >= n_)
      r = n_ - 1;
    if (l > r)
      return;
    int L = l + 1; // convert to 1-based
    int R = r + 1;
    add(B1, L, val);
    add(B1, R + 1, -val);
    add(B2, L, val * T(L - 1));
    add(B2, R + 1, -val * T(R));
  }

  // sum of [0..idx] (0-based). if idx<0 -> 0
  T prefix_sum(int idx) const {
    if (idx < 0)
      return T(0);
    if (idx >= n_)
      idx = n_ - 1;
    int i = idx + 1; // 1-based
    return sum(B1, i) * T(i) - sum(B2, i);
  }

  // sum of [l..r] (0-based)
  T range_sum(int l, int r) const {
    if (l > r)
      return T(0);
    return prefix_sum(r) - prefix_sum(l - 1);
  }

  // value at single index (0-based)
  T point_query(int idx) const { return range_sum(idx, idx); }

  void reset() {
    std::fill(B1.begin(), B1.end(), T(0));
    std::fill(B2.begin(), B2.end(), T(0));
  }

private:
```

```
  int n_;
  vector<T> B1, B2; // 1..n

  static void add(vector<T> &bit, int idx1, T val) {
    int n = (int)bit.size() - 1;
    for (int i = idx1; i <= n; i += (i & -i))
      bit[i] += val;
  }

  static T sum(const vector<T> &bit, int idx1) {
    if (idx1 <= 0)
      return T(0);
    int i = idx1;
    T res = T(0);
    for (; i > 0; i -= (i & -i))
      res += bit[i];
    return res;
  }
};
```

2.6. node_segtree.h

```
struct node {};

node merge_node(const node &a, const node &b) {
  // TODO: implement merging logic for your problem
  return node();
}

struct SegTree {
  int n;
  int off;
  vector<node> st;
  SegTree(const vector<node> &init) {
    n = (int)init.size();
    off = 1;
    while (off < n)
      off <<= 1;
    st.assign(2 * off, node());
    for (int i = 0; i < n; ++i)
      st[off + i] = init[i];
    for (int i = n; i < off; ++i)
      st[off + i] = node(); // empty
    for (int i = off - 1; i >= 1; --i)
      st[i] = merge_node(st[2 * i], st[2 * i + 1]);
  }
  void point_set(int pos, const node &val) {
    int p = off + pos;
    st[p] = val;
    for (p >>= 1; p >= 1; p >>= 1)
      st[p] = merge_node(st[2 * p], st[2 * p + 1]);
  }
  node range_query(int left, int r) {
    if (left > r)
      return node();
    node left_id, right_id;
    int L = left + off, R = r + off;
    while (L <= R) {
      if (L & 1)
        left_id = merge_node(left_id, st[L++]);
      if (!(R & 1))
        right_id = merge_node(st[R--], right_id);
      L >>= 1;
      R >>= 1;
    }
    return merge_node(left_id, right_id);
  }
};

// USAGE NOTES:
// - Implement `node` fields and `merge_node` according to your
operation (sum,
// min, max, gcd, custom struct, etc.).
```

```
// - If you need identity elements, ensure default `node()`
acts as identity or
// replace left_id/right_id initializers.
// - Add #include <bits/stdc++.h> and `using namespace std;` in
file if needed.
```

2.7. range_update_tree.h

```
template <typename T, typename F> struct RangeUpdateTree {
  int n;
  vector<T> tree;
  T identity;
  F merge;
  RangeUpdateTree(const vector<T> &arr, T id, F _m)
      : n((int)arr.size()), tree(2 * n), identity(id),
merge(_m) {
    for (int i = 0; i < n; i++)
      tree[n + i] = arr[i];
    for (int i = n - 1; i >= 1; i--)
      tree[i] = merge(tree[2 * i], tree[2 * i + 1]);
  }
  void update(int l, int r, T value) {
    assert(l >= 0 && r < n && l <= r);
    for (l += n, r += n; l <= r; l >>= 1, r >>= 1) {
      if (l & 1)
        tree[l] = merge(value, tree[l]), l++;
      if (!(r & 1))
        tree[r] = merge(value, tree[r]), r--;
    }
    if (l == r)
      tree[l] = merge(value, tree[l]);
  }
  T query(int v) {
    T res = tree[v += n];
    for (; v > 1; v >>= 1)
      res = merge(res, tree[v >> 1]);
    return res;
  }
};
// ex: RangeUpdateTree<int, decltype(join)> v(vi (n, 1e9), 1e9,
join); use auto
// func for join
```

2.8. sparse_table.h

```
class gcd_sparse_table {
public:
  vector<long long> a;
  int n;
  vector<vector<long long>> sparse_table;

  gcd_sparse_table(vector<long long> &arr) {
    a = arr;
    n = a.size();
    sparse_table.assign(n + 1, vector<long long>(log2(n) + 2));
    build();
  }

  void build() {
    for (int i = 0; i < n; i++)
      sparse_table[i][0] = a[i];
    for (int j = 1; (1 << j) <= n; j++) {
      for (int i = 0; i + (1 << j) - 1 < n; i++) {
        sparse_table[i][j] = __gcd(sparse_table[i][j - 1],
                                   sparse_table[i + (1 << (j -
1))][j - 1]);
      }
    }
  }

  long long query(int l, int r) {
    if (l > r)
      return 0;
```

```
    int len = r - l + 1;
    int p = log2(len);
    return __gcd(sparse_table[l][p], sparse_table[r - (1 << p)
+ 1][p]);
  }
};

class GcdSparseTable2D {
public:
  int n, m, K;
  std::vector<std::vector<long long>> a;
  std::vector<std::vector<std::vector<long long>>> st;

  GcdSparseTable2D(const std::vector<std::vector<long long>>
&matrix) {
    a = matrix;
    n = a.size();
    m = n ? a[0].size() : 0;
    K = std::log2(std::min(n, m)) + 1;
    st.assign(
        n, std::vector<std::vector<long long>>(m,
std::vector<long long>(K)));
    build();
  }

  void build() {
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < m; ++j)
        st[i][j][0] = a[i][j];

    for (int k = 1; k < K; ++k) {
      int len = 1 << k, hl = len >> 1;
      for (int i = 0; i + len <= n; ++i) {
        for (int j = 0; j + len <= m; ++j) {
          long long g1 = std::gcd(st[i][j][k - 1], st[i + hl]
[j][k - 1]);
          long long g2 =
              std::gcd(st[i][j + hl][k - 1], st[i + hl][j + hl]
[k - 1]);
          st[i][j][k] = std::gcd(g1, g2);
        }
      }
    }
  }

  long long query(int x1, int y1, int x2, int y2) {
    if (x1 > x2)
      std::swap(x1, x2);
    if (y1 > y2)
      std::swap(y1, y2);
    int k = std::log2(std::min(x2 - x1 + 1, y2 - y1 + 1));
    int dx = x2 - (1 << k) + 1, dy = y2 - (1 << k) + 1;
    long long t1 = std::gcd(st[x1][y1][k], st[dx][y1][k]);
    long long t2 = std::gcd(st[x1][dy][k], st[dx][dy][k]);
    return std::gcd(t1, t2);
  }
};

class max_sparse_table {
public:
  vector<long long> a;
  int n;
  vector<vector<long long>> sparse_table;

  max_sparse_table(vector<long long> &arr) {
    a = arr;
    n = a.size();
    sparse_table.assign(n + 1, vector<long long>(log2(n) + 2));
    build();
  }

  void build() {
```

```
    for (int i = 0; i < n; i++)
      sparse_table[i][0] = a[i];
    for (int j = 1; (1 << j) <= n; j++) {
      for (int i = 0; i + (1 << j) - 1 < n; i++) {
        sparse_table[i][j] = max(sparse_table[i][j - 1],
                                 sparse_table[i + (1 << (j -
1))][j - 1]);
      }
    }
  }

  long long query(int l, int r) {
    if (l > r)
      return LLONG_MIN;
    int len = r - l + 1;
    int p = log2(len);
    return max(sparse_table[l][p], sparse_table[r - (1 << p) +
1][p]);
  }
};

class MaxSparseTable2D {
public:
  int n, m, K;
  std::vector<std::vector<long long>> a;
  std::vector<std::vector<std::vector<long long>>> st;

  MaxSparseTable2D(const std::vector<std::vector<long long>>
&matrix) {
    a = matrix;
    n = a.size();
    m = n ? a[0].size() : 0;
    K = std::log2(std::min(n, m)) + 1;
    st.assign(
        n, std::vector<std::vector<long long>>(m,
std::vector<long long>(K)));
    build();
  }

  void build() {
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < m; ++j)
        st[i][j][0] = a[i][j];

    for (int k = 1; k < K; ++k) {
      int len = 1 << k, hl = len >> 1;
      for (int i = 0; i + len <= n; ++i) {
        for (int j = 0; j + len <= m; ++j) {
          st[i][j][k] =
              std::max({st[i][j][k - 1], st[i + hl][j][k - 1],
                        st[i][j + hl][k - 1], st[i + hl][j +
hl][k - 1]});
        }
      }
    }
  }

  long long query(int x1, int y1, int x2, int y2) {
    if (x1 > x2)
      std::swap(x1, x2);
    if (y1 > y2)
      std::swap(y1, y2);
    int k = std::log2(std::min(x2 - x1 + 1, y2 - y1 + 1));
    int dx = x2 - (1 << k) + 1, dy = y2 - (1 << k) + 1;
    return std::max(
        {st[x1][y1][k], st[dx][y1][k], st[x1][dy][k], st[dx]
[dy][k]});
  }
};

class min_sparse_table {
public:
```

```
  vector<long long> a;
  int n;
  vector<vector<long long>> sparse_table;

  min_sparse_table(vector<long long> &arr) {
    a = arr;
    n = a.size();
    sparse_table.assign(n + 1, vector<long long>(log2(n) + 2));
    build();
  }

  void build() {
    for (int i = 0; i < n; i++)
      sparse_table[i][0] = a[i];
    for (int j = 1; (1 << j) <= n; j++) {
      for (int i = 0; i + (1 << j) - 1 < n; i++) {
        sparse_table[i][j] = min(sparse_table[i][j - 1],
                                 sparse_table[i + (1 << (j -
1))][j - 1]);
      }
    }
  }

  long long query(int l, int r) {
    if (l > r)
      return LLONG_MAX;
    int len = r - l + 1;
    int p = log2(len);
    return min(sparse_table[l][p], sparse_table[r - (1 << p) +
1][p]);
  }
};

class MinSparseTable2D {
public:
  int n, m, K;
  std::vector<std::vector<long long>> a;
  std::vector<std::vector<std::vector<long long>>> st;

  MinSparseTable2D(const std::vector<std::vector<long long>>
&matrix) {
    a = matrix;
    n = a.size();
    m = n ? a[0].size() : 0;
    K = std::log2(std::min(n, m)) + 1;
    st.assign(
        n, std::vector<std::vector<long long>>(m,
std::vector<long long>(K)));
    build();
  }

  void build() {
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < m; ++j)
        st[i][j][0] = a[i][j];

    for (int k = 1; k < K; ++k) {
      int len = 1 << k, hl = len >> 1;
      for (int i = 0; i + len <= n; ++i) {
        for (int j = 0; j + len <= m; ++j) {
          st[i][j][k] =
              std::min({st[i][j][k - 1], st[i + hl][j][k - 1],
                        st[i][j + hl][k - 1], st[i + hl][j +
hl][k - 1]});
        }
      }
    }
  }

  long long query(int x1, int y1, int x2, int y2) {
    if (x1 > x2)
      std::swap(x1, x2);
```

```
    if (y1 > y2)
      std::swap(y1, y2);
    int k = std::log2(std::min(x2 - x1 + 1, y2 - y1 + 1));
    int dx = x2 - (1 << k) + 1, dy = y2 - (1 << k) + 1;
    return std::min(
        {st[x1][y1][k], st[dx][y1][k], st[x1][dy][k], st[dx]
[dy][k]});
  }
};
```

3. Graph

3.1. DSU.h

```
class DSU {
public:
  vector<int> parent;
  vector<int> size;
  void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
  }
  DSU(int n) {
    parent.resize(n);
    size.resize(n);
    for (int i = 0; i < n; i++) {
      make_set(i);
    }
  }
  int find_set(int v) {
    if (v == parent[v])
      return v;
    return parent[v] = find_set(parent[v]);
  }
  void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
      if (size[a] < size[b])
        swap(a, b);
      parent[b] = a;
      size[a] += size[b];
    }
  }
};
```

3.2. HLD.h

```
struct HLD {
  int n, timer = 0;
  vi top, tin, p, sub;
  HLD(vvi &adj) : n(sz(adj)), top(n), tin(n), p(n, -1), sub(n,
1) {
    vi ord(n + 1);
    for (int i = 0, t = 0, v = ord[i]; i < n; v = ord[++i])
      for (auto &to : adj[v])
        if (to != p[v])
          p[to] = v, ord[++t] = to;
    for (int i = n - 1, v = ord[i]; i > 0; v = ord[--i])
      sub[p[v]] += sub[v];
    for (int v = 0; v < n; v++)
      if (sz(adj[v]))
        iter_swap(begin(adj[v]), max_element(all(adj[v]), [&]
(int a, int b) {
                    return make_pair(a != p[v], sub[a]) <
                           make_pair(b != p[v], sub[b]);
                  }));
    function<void(int)> dfs = [&](int v) {
      tin[v] = timer++;
      for (auto &to : adj[v])
        if (to != p[v]) {
          top[to] = (to == adj[v][0] ? top[v] : to);
          dfs(to);
        }
```

```
    };
    dfs(0);
  }
  int lca(int u, int v) {
    return process(u, v, [](int, int) {});
  }
  template <class B> int process(int a, int b, B op, bool
ignore_lca = false) {
    for (int v;; op(tin[v], tin[b]), b = p[v]) {
      if (tin[a] > tin[b])
        swap(a, b);
      if ((v = top[b]) == top[a])
        break;
    }
    if (int l = tin[a] + ignore_lca, r = tin[b]; l <= r)
      op(l, r);
    return a;
  }
  template <class B> void subtree(int v, B op, bool ignore_lca
= false) {
    if (sub[v] > 1 or !ignore_lca)
      op(tin[v] + ignore_lca, tin[v] + sub[v] - 1);
  }
};
```

3.3. KthAnc.h

```
// O(log n) LCA with Kth anc
struct LCA {
  int n;
  vvi &adjLists;
  int lg;
  vvi up;
  vi depth;
  LCA(vvi &_adjLists, int root = 0) : n(sz(_adjLists)),
adjLists(_adjLists) {
    lg = 1;
    int pw = 1;
    while (pw <= n)
      pw <<= 1, lg++;
    // lg = 20
    up = vvi(n, vi(lg));
    depth.assign(n, -1);
    function<void(int, int)> parentDFS = [&](int from, int
parent) {
      depth[from] = depth[parent] + 1;
      up[from][0] = parent;
      for (auto to : adjLists[from]) {
        if (to == parent)
          continue;
        parentDFS(to, from);
      }
    };
    parentDFS(root, root);
    for (int j = 1; j < lg; j++) {
      for (int i = 0; i < n; i++) {
        up[i][j] = up[up[i][j - 1]][j - 1];
      }
    }
  }
  int kthAnc(int v, int k) {
    int ret = v;
    int pw = 0;
    while (k) {
      if (k & 1)
        ret = up[ret][pw];
      k >>= 1;
      pw++;
    }
    return ret;
  }
  int lca(int u, int v) {
    if (depth[u] > depth[v])
```

```
      swap(u, v);
    v = kthAnc(v, depth[v] - depth[u]);
    if (u == v)
      return v;
    while (up[u][0] != up[v][0]) {
      int i = 0;
      for (; i < lg - 1; i++) {
        if (up[u][i + 1] == up[v][i + 1])
          break;
      }
      u = up[u][i], v = up[v][i];
    }
    return up[u][0];
  };
  int dist(int u, int v) { return depth[u] + depth[v] - 2 *
depth[lca(u, v)]; }
};
```

3.4. LCA.h

```
// O(1) LCA
struct LCA {
  int T = 0;
  vi st, path, ret;
  vi en, d;
  RMQ<int> rmq;
  LCA(vector<vi> &C)
      : st(sz(C)), en(sz(C)), d(sz(C)), rmq((dfs(C, 0, -1),
ret)) {}
  void dfs(vvi &adj, int v, int par) {
    st[v] = T++;
    for (auto to : adj[v])
      if (to != par) {
        path.pb(v), ret.pb(st[v]);
        d[to] = d[v] + 1;
        dfs(adj, to, v);
      }
    en[v] = T - 1;
  }
  bool anc(int p, int c) { return st[p] <= st[c] and en[p] >=
en[c]; }
  int lca(int a, int b) {
    if (a == b)
      return a;
    tie(a, b) = minmax(st[a], st[b]);
    return path[rmq.query(a, b - 1)];
  }
  int dist(int a, int b) { return d[a] + d[b] - 2 * d[lca(a,
b)]; }
};
```

3.5. bellman.h

```
bool bellman_ford(int n, int src, vector<vector<pair<int,
int>>> &adj,
                  vector<long long> &dist) {
  const long long INF = 1e18;
  dist.assign(n, INF);
  dist[src] = 0;
  for (int i = 0; i < n - 1; i++) {
    for (int u = 0; u < n; u++) {
      if (dist[u] == INF)
        continue;
      for (auto &p : adj[u]) {
        int v = p.first, w = p.second;
        if (dist[u] + w < dist[v])
          dist[v] = dist[u] + w;
      }
    }
  }
  for (int u = 0; u < n; u++) {
    if (dist[u] == INF)
      continue;
    for (auto &p : adj[u]) {
```

```
      int v = p.first, w = p.second;
      if (dist[u] + w < dist[v])
        return false;
    }
  }
  return true;
}
```

3.6. blockcuttree.h

```
struct BlockCutTree {
  int n;
  vector<vector<int>> adj;
  vector<int> tin, low;
  int timer;
  stack<int> st;
  vector<vector<int>> tree_adj;
  vector<int> is_ap;
  int num_blocks;

  BlockCutTree(int n)
      : n(n), adj(n), tin(n, -1), low(n, -1), timer(0),
is_ap(n, 0),
        num_blocks(0) {}

  void add_edge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  void dfs(int u, int p = -1) {
    tin[u] = low[u] = timer++;
    st.push(u);
    int children = 0;
    for (int v : adj[u]) {
      if (v == p)
        continue;
      if (tin[v] != -1) {
        low[u] = min(low[u], tin[v]);
      } else {
        dfs(v, u);
        low[u] = min(low[u], low[v]);
        if (low[v] >= tin[u] && p != -1)
          is_ap[u] = 1;
        children++;
        if (low[v] >= tin[u]) {
          int block_node = n + num_blocks++;
          tree_adj.emplace_back();
          tree_adj[block_node].push_back(u);
          tree_adj[u].push_back(block_node);
          while (true) {
            int node = st.top();
            st.pop();
            if (node != u) {
              tree_adj[block_node].push_back(node);
              tree_adj[node].push_back(block_node);
            }
            if (node == v)
              break;
          }
        }
      }
    }
    if (p == -1 && children > 1)
      is_ap[u] = 1;
  }

  void build() {
    timer = num_blocks = 0;
    tree_adj.assign(n, {});
    fill(tin.begin(), tin.end(), -1);
    fill(is_ap.begin(), is_ap.end(), 0);
    while (!st.empty())
```

```
      st.pop();
    for (int i = 0; i < n; ++i) {
      if (tin[i] == -1) {
        dfs(i);
        while (!st.empty())
          st.pop();
      }
    }
  }

  int size() const { return n + num_blocks; }
};
```

3.7. bridges.h

```
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);
  }
}

// ARTICULATION POINTS:
int n;
vector<vector<int>> adj;
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);
  }
}

// arya bridges
void findBridges_dfs(int u, int p, int &time,
vector<vector<int>> &adj,
                     vector<int> &disc, vector<int> &low,
                     vector<pair<int, int>> &bridges) {
  disc[u] = low[u] = time++;

  for (int v : adj[u]) {
    if (v == p)
      continue;

    if (disc[v] != -1) {
      low[u] = min(low[u], disc[v]);
    } else {
      findBridges_dfs(v, u, time, adj, disc, low, bridges);
      low[u] = min(low[u], low[v]);
      if (low[v] > disc[u]) {
        bridges.push_back({u, v});
      }
    }
  }
}
```

```
vector<pair<int, int>> findBridges(int n, vector<vector<int>>
&adj) {
  vector<int> disc(n, -1), low(n, -1);
  vector<pair<int, int>> bridges;
  int time = 0;

  for (int i = 0; i < n; ++i) {
    if (disc[i] == -1) {
      findBridges_dfs(i, -1, time, adj, disc, low, bridges);
    }
  }
  return bridges;
}
```

3.8. dijkstra.h

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> &d, vector<int> &p) {
  int n = adj.size();
  d.assign(n, INF);
  p.assign(n, -1);
  vector<bool> u(n, false);

  d[s] = 0;
  for (int i = 0; i < n; i++) {
    int v = -1;
    for (int j = 0; j < n; j++) {
      if (!u[j] && (v == -1 || d[j] < d[v]))
        v = j;
    }

    if (d[v] == INF)
      break;
```

```
    u[v] = true;
    for (auto edge : adj[v]) {
      int to = edge.first;
      int len = edge.second;

      if (d[v] + len < d[to]) {
        d[to] = d[v] + len;
        p[to] = v;
      }
    }
  }
}
```

3.9. floyd_washall.h

```
const long long INF = (long long)1e18;
bool floyd_warshall(int n, vector<vector<long long>> &dist) {
  // initialise dist with edge weights, INF if no edge exists
  for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        if (dist[i][k] < INF && dist[k][j] < INF)
          dist[i][j] = min(dist[i][j], dist[i][k] + dist[k]
[j]);

  for (int i = 0; i < n; i++)
    if (dist[i][i] < 0)
      return true;
  return false;
}
```

4. Number Theory

4.1. ModularArithmetic.h

```
int add(int x, int y, int m = M) {
  int ret = (x + y) % m;
  if (ret < 0)
    ret += m;
  return ret;
}
int mult(int x, int y, int m = M) {
  int ret = (x * y) % m;
  if (ret < 0)
    ret += m;
  return ret;
}
int pw(int a, int b, int m = M) {
  int ret = 1;
  int p = a;
  while (b) {
    if (b & 1)
      ret = mult(ret, p, m);
    b >>= 1;
    p = mult(p, p, m);
  }
  return ret;
}

#define LL int
const long long mod = 1e9 + 7;

int euclid(int a, int b, int &x, int &y) {
  if (!b)
    return x = 1, y = 0, a;
  int d = euclid(b, a % b, y, x);
  return y -= a / b * x, d;
}

int modulo_inverse(int a, int m) {
  int x, y;
  int g = euclid(a, m, x, y);
  if (g != 1) {
    return -1;
```

```
  } else {
    x = (x % m + m) % m;
    return x;
  }
}

LL mod_mul(LL a, LL b) {
  a = a % mod;
  b = b % mod;
  return (((a * b) % mod) + mod) % mod;
}

LL mod_add(LL a, LL b) {
  a = a % mod;
  b = b % mod;
  return (((a + b) % mod) + mod) % mod;
}

const int MX = 5e5 + 1;
vector<int> inv(MX + 1), fci(MX + 1), fc(MX + 1);
const int Mod = 1e9 + 7;

void Inverses() {
  inv[1] = 1;
  for (int i = 2; i <= MX; i++) {
    inv[i] = Mod - Mod / i * inv[Mod % i] % Mod;
  }
}

void Factorials() {
  fc[0] = fc[1] = 1;
  for (int i = 2; i <= MX; i++) {
    fc[i] = fc[i - 1] * i % Mod;
  }
}

void InverseFactorials() {
  Inverses();
  Factorials();
  fci[1] = fci[0] = 1;
  for (int i = 2; i <= MX; i++) {
    fci[i] = fci[i - 1] * inv[i] % Mod;
  }
}

int nck(int num, int k) {
  if (num < 0) {
    return 0;
  }
  if (k < 0) {
    return 0;
  }
  if (num < k) {
    return 0;
  } else {
    return fc[num] * fci[k] % Mod * fci[num - k] % Mod;
  }
}

int BinExpItermod(int a, int b) {
  int ans = 1;
  while (b > 0) {
    if (b & 1) {
      ans = (ans * a) % mod;
    }
    a = (a * a) % mod;
    b = b >> 1;
  }
  return ans;
}
```

4.2. bit_bns.h

```
// --- Bit Binary Search in o(log(n)) ---
const int M = 20 const int N = 1 << M

                              int
                              lower_bound(int val) {
  int ans = 0, sum = 0;
  for (int i = M - 1; i >= 0; i--) {
    int x = ans + (1 << i);
    if (sum + bit[x] < val)
      ans = x, sum += bit[x];
  }

  return ans + 1;
}
```

4.3. crt.h

```
#ifndef CRT_H
#define CRT_H

#include <numeric>
#include <utility>
#include <vector>

namespace CRT {
using ll = long long;
using u128 = __int128;

inline ll mult(ll a, ll b, ll mod) { return (ll)((u128)a * b %
mod); }

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

// Solves x = a_i (mod m_i). Returns {x, LCM(m_i)}.
// If no solution, returns {-1, -1}.
// Works for non-coprime moduli.
std::pair<ll, ll> crt(const std::vector<ll> &a, const
std::vector<ll> &m) {
  ll x = 0, M = 1;
  if (a.empty())
    return {0, 1};
  x = a[0];
  M = m[0];
  for (size_t i = 1; i < a.size(); ++i) {
    ll a_i = a[i], m_i = m[i];
    ll c = (a_i - x % m_i + m_i) % m_i;
    ll x_i, y_i;
    ll g = exEuclid(M, m_i, x_i, y_i);
    if (c % g != 0)
      return {-1, -1}; // No solution
    ll k = mult(c / g, x_i, m_i / g);
    x = x + k * M;
    M = std::lcm(M, m_i);
    x = (x % M + M) % M;
  }
  return {x, M};
}
} // namespace CRT
#endif
```

4.4. discretelog.h

```
#ifndef DISCRETE_LOG_H
#define DISCRETE_LOG_H

#include <cmath>
#include <map>
#include <numeric>

namespace DiscreteLog {
using ll = long long;
using u128 = __int128;

inline ll mult(ll a, ll b, ll mod) { return (ll)((u128)a * b %
mod); }

ll power(ll a, ll b, ll m) {
  ll res = 1;
  a %= m;
  while (b > 0) {
    if (b & 1)
      res = mult(res, a, m);
    a = mult(a, a, m);
    b >>= 1;
  }
  return res;
}

// Solves a^x = b (mod m). Returns x, or -1 if no solution.
// Assumes gcd(a, m) = 1.
ll bsgs(ll a, ll b, ll m) {
  a %= m;
  b %= m;
  if (b == 1 || m == 1)
    return 0;
  ll n = (ll)std::sqrt(m) + 1;
  std::map<ll, ll> giant_steps;
  ll an = 1;
  for (ll i = 0; i < n; i++)
    an = mult(an, a, m);

  for (ll q = 0, cur = b; q <= n; q++) {
    giant_steps[cur] = q;
    cur = mult(cur, a, m);
  }

  for (ll p = 1, cur = 1; p <= n; p++) {
    cur = mult(cur, an, m);
    if (giant_steps.count(cur)) {
      ll x = p * n - giant_steps[cur];
      if (power(a, x, m) == b)
        return x;
    }
  }
  return -1;
}
} // namespace DiscreteLog
#endif
```

4.5. discroot.h

```
#ifndef DISCRETE_ROOT_H
#define DISCRETE_ROOT_H

#include <algorithm>
#include <array>
#include <cmath>
#include <map>
#include <numeric>
#include <vector>

namespace DiscreteRoot {
using ll = long long;
using u128 = __int128;
```

```cpp
// --- Core Dependencies ---
inline ll mult(ll a, ll b, ll mod) { return (ll)((u128)a * b %
mod); }

ll power(ll a, ll b, ll m) {
  ll res = 1;
  a %= m;
  while (b > 0) {
    if (b & 1)
      res = mult(res, a, m);
    a = mult(a, a, m);
    b >>= 1;
  }
  return res;
}

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

// --- Primality & Factorization Dependencies ---
bool miller_rabin(ll n, int a, ll d, int s) {
  ll x = power(a, d, n);
  if (x == 1 || x == n - 1)
    return true;
  for (int r = 1; r < s; r++) {
    x = mult(x, x, n);
    if (x == n - 1)
      return true;
  }
  return false;
}

bool is_prime(ll n) {
  if (n < 2 || (n % 2 == 0 && n > 2))
    return false;
  if (n == 2 || n == 3)
    return true;
  ll d = n - 1;
  int s = 0;
  while ((d & 1) == 0) {
    d >>= 1;
    s++;
  }
  static const std::array<int, 12> bases = {2,  3,  5,  7,  11,
13,
                                            17, 19, 23, 29, 31,
37};
  for (int a : bases) {
    if (n <= a)
      break;
    if (!miller_rabin(n, a, d, s))
      return false;
  }
  return true;
}

ll brent(ll n, ll x0 = 2, ll c = 1) {
  ll x = x0;
  ll g = 1;
  ll q = 1;
  ll xs, y;
```

```
  int m = 128;
  int l = 1;
  while (g == 1) {
    y = x;
    for (int i = 1; i < l; i++)
      x = (mult(x, x, n) + c) % n;
    int k = 0;
    while (k < l && g == 1) {
      xs = x;
      for (int i = 0; i < m && i < l - k; i++) {
        x = (mult(x, x, n) + c) % n;
        q = mult(q, std::abs(y - x), n);
      }
      g = std::gcd(q, n);
      k += m;
    }
    l *= 2;
  }
  if (g == n) {
    do {
      xs = (mult(xs, xs, n) + c) % n;
      g = std::gcd(std::abs(xs - y), n);
    } while (g == 1);
  }
  return g;
}

void factorize_recursive(ll n, std::vector<ll> &factors) {
  if (n == 1)
    return;
  if (is_prime(n)) {
    factors.push_back(n);
    return;
  }
  ll g = n;
  for (ll c = 1; g == n; c++) {
    g = brent(n, 2, c);
  }
  factorize_recursive(g, factors);
  factorize_recursive(n / g, factors);
}

std::vector<ll> factorize(ll n) {
  std::vector<ll> factors;
  factorize_recursive(n, factors);
  std::sort(factors.begin(), factors.end());
  return factors;
}

// --- Primitive Root Dependency ---
ll primitive_root(ll p) {
  if (!is_prime(p))
    return -1;
  ll phi = p - 1;
  std::vector<ll> factors = factorize(phi);
  factors.erase(std::unique(factors.begin(), factors.end()),
factors.end());
  for (ll g = 2; g <= p; g++) {
    bool ok = true;
    for (ll f : factors) {
      if (power(g, phi / f, p) == 1) {
        ok = false;
        break;
      }
    }
    if (ok)
      return g;
  }
  return -1;
}

// --- Discrete Log Dependency ---
```

```
ll bsgs(ll a, ll b, ll m) {
  a %= m;
  b %= m;
  if (b == 1 || m == 1)
    return 0;
  ll n = (ll)std::sqrt(m) + 1;
  std::map<ll, ll> giant_steps;
  ll an = 1;
  for (ll i = 0; i < n; i++)
    an = mult(an, a, m);
  for (ll q = 0, cur = b; q <= n; q++) {
    giant_steps[cur] = q;
    cur = mult(cur, a, m);
  }
  for (ll p = 1, cur = 1; p <= n; p++) {
    cur = mult(cur, an, m);
    if (giant_steps.count(cur)) {
      ll x = p * n - giant_steps[cur];
      if (power(a, x, m) == b)
        return x;
    }
  }
  return -1;
}

// --- Discrete Root ---
// Solves x^k = a (mod p) for prime p.
ll discrete_root(ll k, ll a, ll p) {
  if (a == 0)
    return 0;
  ll g = primitive_root(p);
  if (g == -1)
    return -1; // Not prime
  ll b = bsgs(g, a, p);
  if (b == -1)
    return -1; // a is not in the group
  ll x_0, y_0;
  ll d = exEuclid(k, p - 1, x_0, y_0);
  if (b % d != 0)
    return -1;
  ll x = mult(x_0, b / d, (p - 1) / d);
  x = (x % ((p - 1) / d) + ((p - 1) / d)) % ((p - 1) / d);
  return power(g, x, p);
}
} // namespace DiscreteRoot
#endif
```

4.6. euler_totient_fxn.h

```
int euler_totient_function(int n) {
  int res = 1;
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
      int count = 0;
      int val = 1;
      while (n % i == 0) {
        count++;
        val = val * i;
        n = n / i;
      }
      res = res * ((val / i) * (i - 1));
    }
  }
  if (n > 1) {
    res = res * (n - 1);
  }
  return res;
}
```

4.7. factorization.h

```
namespace NT {
using ll = long long;
using u128 = __int128;
```

```
inline ll mult(ll a, ll b, ll mod) { return (ll)((u128)a * b %
mod); }

ll power(ll a, ll b, ll m) {
  ll res = 1;
  a %= m;
  while (b > 0) {
    if (b & 1)
      res = mult(res, a, m);
    a = mult(a, a, m);
    b >>= 1;
  }
  return res;
}

bool miller_rabin(ll n, int a, ll d, int s) {
  ll x = power(a, d, n);
  if (x == 1 || x == n - 1)
    return true;
  for (int r = 1; r < s; r++) {
    x = mult(x, x, n);
    if (x == n - 1)
      return true;
  }
  return false;
}

// Deterministic Miller-Rabin for 64-bit integers
bool is_prime(ll n) {
  if (n < 2)
    return false;
  if (n == 2 || n == 3)
    return true;
  if (n % 2 == 0 || n % 3 == 0)
    return false;
  ll d = n - 1;
  int s = 0;
  while ((d & 1) == 0) {
    d >>= 1;
    s++;
  }
  static const array<int, 12> bases = {2,  3,  5,  7,  11, 13,
                                       17, 19, 23, 29, 31, 37};
  for (int a : bases) {
    if (n <= a)
      break;
    if (!miller_rabin(n, a, d, s))
      return false;
  }
  return true;
}

ll brent(ll n, ll x0 = 2, ll c = 1) {
  ll x = x0;
  ll g = 1;
  ll q = 1;
  ll xs, y;
  int m = 128;
  int l = 1;
  while (g == 1) {
    y = x;
    for (int i = 1; i < l; i++)
      x = (mult(x, x, n) + c) % n;
    int k = 0;
    while (k < l && g == 1) {
      xs = x;
      for (int i = 0; i < m && i < l - k; i++) {
        x = (mult(x, x, n) + c) % n;
        q = mult(q, abs(y - x), n);
      }
      g = gcd(q, n);
```

```
      k += m;
    }
    l *= 2;
  }
  if (g == n) {
    do {
      xs = (mult(xs, xs, n) + c) % n;
      g = gcd(abs(xs - y), n);
    } while (g == 1);
  }
  return g;
}

void factorize_recursive(ll n, vector<ll> &factors) {
  if (n == 1)
    return;
  // Trial division for small primes can speed up standard
cases
  if (n < 1000) {
    for (ll p : {2, 3, 5}) {
      while (n % p == 0) {
        factors.push_back(p);
        n /= p;
      }
    }
    if (n == 1)
      return;
  }
  if (is_prime(n)) {
    factors.push_back(n);
    return;
  }
  ll g = n;
  // Retry brent with different params if it fails
  for (ll c = 1; g == n; c++) {
    g = brent(n, 2, c);
  }
  factorize_recursive(g, factors);
  factorize_recursive(n / g, factors);
}

// Main function to call
vector<ll> factorize(ll n) {
  vector<ll> factors;
  factorize_recursive(n, factors);
  sort(factors.begin(), factors.end());
  return factors;
}
} // namespace NT
```

4.8. garner.h

```
#ifndef GARNER_H
#define GARNER_H

#include <numeric>
#include <vector>

namespace Garner {
using ll = long long;
using u128 = __int128;

inline ll mult(ll a, ll b, ll mod) { return (ll)((u128)a * b %
mod); }

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

```
  x = y1;
  y = x1 - y1 * (a / b);
  return d;
}

ll modInverse(ll a, ll m) {
  ll x, y;
  ll g = exEuclid(a, m, x, y);
  if (g != 1)
    return -1;
  return (x % m + m) % m;
}

// Solves x = a_i (mod m_i) for *coprime* m_i.
ll garner(const std::vector<ll> &a, const std::vector<ll> &m) {
  int n = a.size();
  std::vector<ll> x(n);
  for (int i = 0; i < n; ++i) {
    x[i] = a[i];
    for (int j = 0; j < i; ++j) {
      ll inv = modInverse(m[j], m[i]);
      x[i] = mult((x[i] - x[j] + m[i]), inv, m[i]);
    }
  }
  ll res = x[n - 1];
  // Use a large prime (e.g., 10^18 + 9) or __int128 for M if
result overflows
  // ll
  ll M = 1e18 + 9;
  for (int i = n - 2; i >= 0; --i) {
    res = (mult(res, m[i], M) + x[i]) % M;
  }
  return res;
}
} // namespace Garner
#endif
```

4.9. matrix_expo.h

```
int **matrixmul(int **matrix1, int **matrix2) {
  int **matrix3 = new int *[2];
  for (int i = 0; i < 2; i++)
    matrix3[i] = new int[2];

  matrix3[0][0] =
      (matrix1[0][0] * matrix2[0][0]) + (matrix1[0][1] *
matrix2[1][0]);
  matrix3[0][1] =
      (matrix1[0][0] * matrix2[0][1]) + (matrix1[0][1] *
matrix2[1][1]);
  matrix3[1][0] =
      (matrix1[1][0] * matrix2[0][0]) + (matrix1[1][1] *
matrix2[1][0]);
  matrix3[1][1] =
      (matrix1[1][0] * matrix2[0][1]) + (matrix1[1][1] *
matrix2[1][1]);
  matrix3[0][0] %= M;
  matrix3[1][0] %= M;
  matrix3[0][1] %= M;
  matrix3[1][1] %= M;

  return matrix3;
}

int **matrixexpo(int **matrix, int n, int **ans) {
  while (n > 0) {
    if (n % 2 == 1)
      ans = matrixmul(ans, matrix);
    matrix = matrixmul(matrix, matrix);
    n /= 2;
  }
  return ans;
}
```

4.10. montgomery.h

4.11. precompute_divisors.h

```
const int MAX = 1e5;
vector<int> divisor[MAX + 1];
void sieve() {
  for (int i = 1; i <= MAX; ++i) {
    for (int j = i; j <= MAX; j += i)
      divisor[j].push_back(i);
  }
}
```

4.12. primroot.h

```
#ifndef PRIMITIVE_ROOT_H
#define PRIMITIVE_ROOT_H

#include <algorithm>
#include <array>
#include <cmath>
#include <numeric>
#include <vector>

namespace PrimitiveRoot {
using ll = long long;
using u128 = __int128;

inline ll mult(ll a, ll b, ll mod) { return (ll)((u128)a * b %
mod); }

ll power(ll a, ll b, ll m) {
  ll res = 1;
  a %= m;
  while (b > 0) {
    if (b & 1)
      res = mult(res, a, m);
    a = mult(a, a, m);
    b >>= 1;
  }
  return res;
}

bool miller_rabin(ll n, int a, ll d, int s) {
  ll x = power(a, d, n);
  if (x == 1 || x == n - 1)
    return true;
  for (int r = 1; r < s; r++) {
    x = mult(x, x, n);
    if (x == n - 1)
      return true;
  }
  return false;
}

bool is_prime(ll n) {
  if (n < 2 || (n % 2 == 0 && n > 2))
    return false;
  if (n == 2 || n == 3)
    return true;
  ll d = n - 1;
  int s = 0;
  while ((d & 1) == 0) {
    d >>= 1;
    s++;
  }
  static const std::array<int, 12> bases = {2,  3,  5,  7,  11,
13,
                                            17, 19, 23, 29, 31,
37};
  for (int a : bases) {
    if (n <= a)
```

```
      break;
    if (!miller_rabin(n, a, d, s))
      return false;
  }
  return true;
}

ll brent(ll n, ll x0 = 2, ll c = 1) {
  ll x = x0;
  ll g = 1;
  ll q = 1;
  ll xs, y;
  int m = 128;
  int l = 1;
  while (g == 1) {
    y = x;
    for (int i = 1; i < l; i++)
      x = (mult(x, x, n) + c) % n;
    int k = 0;
    while (k < l && g == 1) {
      xs = x;
      for (int i = 0; i < m && i < l - k; i++) {
        x = (mult(x, x, n) + c) % n;
        q = mult(q, std::abs(y - x), n);
      }
      g = std::gcd(q, n);
      k += m;
    }
    l *= 2;
  }
  if (g == n) {
    do {
      xs = (mult(xs, xs, n) + c) % n;
      g = std::gcd(std::abs(xs - y), n);
    } while (g == 1);
  }
  return g;
}

void factorize_recursive(ll n, std::vector<ll> &factors) {
  if (n == 1)
    return;
  if (is_prime(n)) {
    factors.push_back(n);
    return;
  }
  ll g = n;
  for (ll c = 1; g == n; c++) {
    g = brent(n, 2, c);
  }
  factorize_recursive(g, factors);
  factorize_recursive(n / g, factors);
}

std::vector<ll> factorize(ll n) {
  std::vector<ll> factors;
  factorize_recursive(n, factors);
  std::sort(factors.begin(), factors.end());
  return factors;
}

// Finds the smallest primitive root modulo p.
ll primitive_root(ll p) {
  if (!is_prime(p))
    return -1;
  ll phi = p - 1;
  std::vector<ll> factors = factorize(phi);
  factors.erase(std::unique(factors.begin(), factors.end()),
factors.end());
  for (ll g = 2; g <= p; g++) {
    bool ok = true;
    for (ll f : factors) {
```

```
      if (power(g, phi / f, p) == 1) {
        ok = false;
        break;
      }
    }
    if (ok)
      return g;
  }
  return -1;
}
} // namespace PrimitiveRoot
#endif
```

4.13. some_dp.h

```
// LIS
int lis(vector<int> const &a) {
  int n = a.size();
  const int INF = 1e9;
  vector<int> d(n + 1, INF);
  d[0] = -INF;

  for (int i = 0; i < n; i++) {
    int l = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
    if (d[l - 1] < a[i] && a[i] < d[l])
      d[l] = a[i];
  }

  int ans = 0;
  for (int l = 0; l <= n; l++) {
    if (d[l] < INF)
      ans = l;
  }
  return ans;
}
// or segtree lol
```

4.14. spf.h

```
int MX = 1e7 + 1;
vi spf(MX + 1, INT32_MAX);
vector<int> is_prime(MX + 1, 1);
void sieve(int n = MX) {
  is_prime[0] = is_prime[1] = 0;
  int cnt = 1;
  for (int i = 2; i <= n; i++) {
    if (is_prime[i]) {
      for (int j = i * i; j <= n; j += i) {
        is_prime[j] = 0;
        spf[j] = min(i, spf[j]);
      }
      is_prime[i] = cnt;
      cnt++;
    }
  }
  return;
}
```

5. Strings

5.1. Trie.h

```
class TrieNode {
public:
  unordered_map<char, TrieNode *> children;
  bool isEndOfWord;

  TrieNode() : isEndOfWord(false) {}
};
class Trie {
private:
  TrieNode *root;

public:
  Trie() { root = new TrieNode(); }
```

```
  void insert(const string &word) {
    TrieNode *node = root;
    for (char ch : word) {
      if (node->children.find(ch) == node->children.end()) {
        node->children[ch] = new TrieNode();
      }
      node = node->children[ch];
    }
    node->isEndOfWord = true;
  }
  bool search(const string &word) {
    TrieNode *node = root;
    for (char ch : word) {
      if (node->children.find(ch) == node->children.end()) {
        return false;
      }
      node = node->children[ch];
    }
    return node->isEndOfWord;
  }
  bool startsWith(const string &prefix) {
    TrieNode *node = root;
    for (char ch : prefix) {
      if (node->children.find(ch) == node->children.end()) {
        return false;
      }
      node = node->children[ch];
    }
    return true;
  }
};
```

5.2. ahocorasick.h

```
class AhoCorasick {
public:
  static const int K = 26;

  struct Vertex {
    int next[K];
    int p = -1;
    char pch;
    int link = -1;
    int go[K];
    vector<int> out; // pattern indices that end here

    Vertex(int p = -1, char ch = '$') : p(p), pch(ch) {
      fill(begin(next), end(next), -1);
      fill(begin(go), end(go), -1);
    }
  };

  vector<Vertex> t;
  int pattern_count = 0; // next pattern id to assign

  AhoCorasick() {
    t.emplace_back(); // root vertex (index 0)
  }

  // add a lowercase 'a'..'z' pattern, returns its pattern
index (0-based)
  int add_string(const string &s) {
    int v = 0;
    for (char ch : s) {
      int c = ch - 'a';
      if (c < 0 || c >= K)
        throw runtime_error("patterns must be lowercase a-z");
      if (t[v].next[c] == -1) {
        t[v].next[c] = (int)t.size();
        t.emplace_back(v, ch);
      }
      v = t[v].next[c];
    }
```

```
    int id = pattern_count++;
    t[v].out.push_back(id);
    return id;
  }

  // returns vector of (text_index, patternIndex) pairs for
every match
  vector<pair<int, int>> search(const string &text) {
    vector<pair<int, int>> res;
    int v = 0;
    for (int i = 0; i < (int)text.size(); ++i) {
      char ch = text[i];
      if (ch < 'a' || ch > 'z') {
        // reset on invalid char (same behaviour as before)
        v = 0;
        continue;
      }
      v = go(v, ch);

      // follow output/link chain to collect matches
      for (int u = v; u != 0; u = get_link(u)) {
        for (int patId : t[u].out) {
          res.emplace_back(i, patId);
        }
      }
      // optionally check root outputs (only relevant if you
inserted empty
      // pattern)
      for (int patId : t[0].out) {
        res.emplace_back(i, patId);
      }
    }
    return res;
  }

  int get_link(int v) {
    if (t[v].link == -1) {
      if (v == 0 || t[v].p == 0)
        t[v].link = 0;
      else
        t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
  }

  int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
      if (t[v].next[c] != -1)
        t[v].go[c] = t[v].next[c];
      else
        t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
  }
};
```

5.3. expression_parsing.h

```
bool delim(char c) { return c == ' '; }

bool is_op(char c) { return c == '+' || c == '-' || c == '*' ||
c == '/'; }

bool is_unary(char c) { return c == '+' || c == '-'; }

int priority(char op) {
  if (op < 0) // unary operator
    return 3;
  if (op == '+' || op == '-')
    return 1;
  if (op == '*' || op == '/')
    return 2;
```

```
  return -1;
}

void process_op(stack<int> &st, char op) {
  if (op < 0) {
    int l = st.top();
    st.pop();
    switch (-op) {
    case '+':
      st.push(l);
      break;
    case '-':
      st.push(-l);
      break;
    }
  } else {
    int r = st.top();
    st.pop();
    int l = st.top();
    st.pop();
    switch (op) {
    case '+':
      st.push(l + r);
      break;
    case '-':
      st.push(l - r);
      break;
    case '*':
      st.push(l * r);
      break;
    case '/':
      st.push(l / r);
      break;
    }
  }
}

int evaluate(string &s) {
  stack<int> st;
  stack<char> op;
  bool may_be_unary = true;
  for (int i = 0; i < (int)s.size(); i++) {
    if (delim(s[i]))
      continue;

    if (s[i] == '(') {
      op.push('(');
      may_be_unary = true;
    } else if (s[i] == ')') {
      while (op.top() != '(') {
        process_op(st, op.top());
        op.pop();
      }
      op.pop();
      may_be_unary = false;
    } else if (is_op(s[i])) {
      char cur_op = s[i];
      if (may_be_unary && is_unary(cur_op))
        cur_op = -cur_op;
      while (!op.empty() &&
             ((cur_op >= 0 && priority(op.top()) >=
priority(cur_op)) ||
              (cur_op < 0 && priority(op.top()) >
priority(cur_op)))) {
        process_op(st, op.top());
        op.pop();
      }
      op.push(cur_op);
      may_be_unary = true;
    } else {
      int number = 0;
      while (i < (int)s.size() && isalnum(s[i]))
```

```
        number = number * 10 + s[i++] - '0';
      --i;
      st.push(number);
      may_be_unary = false;
    }
  }

  while (!op.empty()) {
    process_op(st, op.top());
    op.pop();
  }
  return st.top();
}
```

5.4. finding_repetitions.h

```
vector<int> z_function(string const &s) {
  int n = s.size();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; i++) {
    if (i <= r)
      z[i] = 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;
}

int get_z(vector<int> const &z, int i) {
  if (0 <= i && i < (int)z.size())
    return z[i];
  else
    return 0;
}

vector<pair<int, int>> repetitions;

void convert_to_repetitions(int shift, bool left, int cntr, int
l, int k1,
                            int k2) {
  for (int l1 = max(1, l - k2); l1 <= min(l, k1); l1++) {
    if (left && l1 == l)
      break;
    int l2 = l - l1;
    int pos = shift + (left ? cntr - l1 : cntr - l - l1 + 1);
    repetitions.emplace_back(pos, pos + 2 * l - 1);
  }
}

void find_repetitions(string s, int shift = 0) {
  int n = s.size();
  if (n == 1)
    return;

  int nu = n / 2;
  int nv = n - nu;
  string u = s.substr(0, nu);
  string v = s.substr(nu);
  string ru(u.rbegin(), u.rend());
  string rv(v.rbegin(), v.rend());

  find_repetitions(u, shift);
  find_repetitions(v, shift + nu);

  vector<int> z1 = z_function(ru);
  vector<int> z2 = z_function(v + '#' + u);
  vector<int> z3 = z_function(ru + '#' + rv);
  vector<int> z4 = z_function(v);
```

```
  for (int cntr = 0; cntr < n; cntr++) {
    int l, k1, k2;
    if (cntr < nu) {
      l = nu - cntr;
      k1 = get_z(z1, nu - cntr);
      k2 = get_z(z2, nv + 1 + cntr);
    } else {
      l = cntr - nu + 1;
      k1 = get_z(z3, nu + 1 + nv - 1 - (cntr - nu));
      k2 = get_z(z4, (cntr - nu) + 1);
    }
    if (k1 + k2 >= l)
      convert_to_repetitions(shift, cntr < nu, cntr, l, k1,
k2);
  }
}
```

5.5. hash.h

```
template <int MOD, int P> struct RH {
  // using H1 = RH<1000000007, 91138233>;
  // using H2 = RH<1000000009, 97266353>;
  vector<long long> h, p;
  RH(const string &s) {
    int n = s.size();
    h.resize(n + 1, 0);
    p.resize(n + 1, 0);
    p[0] = 1;
    for (int i = 0; i < n; i++) {
      h[i + 1] = (h[i] * P + s[i]) % MOD;
      p[i + 1] = p[i] * P % MOD;
    }
  }
  long long get(int l, int r) { // [l,r]
    long long res = (h[r + 1] - h[l] * p[r - l + 1]) % MOD;
    return res < 0 ? res + MOD : res;
  }
};
```

5.6. kmp.h

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

vector<int> KMP(string text, string pattern) {
  string s = pattern + "#" + text;
  vector<int> pi = prefix_function(s);
  vector<int> matches;
  int p = pattern.length();

  for (int i = 0; i < s.length(); i++) {
    if (pi[i] == p) {
      int match_pos = i - 2 * p;
      matches.push_back(match_pos); // 0-based index in text
    }
  }
  return matches;
}
```

5.7. longest_common_prefix.h

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

```cpp
// --- suffix array (sort_cyclic_shifts + suffix_array) ---
vector<int> sort_cyclic_shifts(const string &a) {
  int n = a.size();
  const int alphabet = 256;

  vector<int> p(n), c(n);
  vector<int> cnt(max(alphabet, n), 0);

  for (int i = 0; i < n; i++)
    cnt[(unsigned char)a[i]]++;
  for (int i = 1; i < alphabet; i++)
    cnt[i] = cnt[i - 1] + cnt[i];
  for (int i = n - 1; i >= 0; i--)
    p[--cnt[(unsigned char)a[i]]] = i;

  c[p[0]] = 0;
  int classes = 1;
  for (int i = 1; i < n; i++) {
    if (a[p[i]] != a[p[i - 1]])
      classes++;
    c[p[i]] = classes - 1;
  }

  for (int h = 0; (1 << h) < n; h++) {
    vector<int> pn(n), cn(n);
    for (int i = 0; i < n; i++) {
      pn[i] = p[i] - (1 << h);
      if (pn[i] < 0)
        pn[i] += n;
    }
    fill(cnt.begin(), cnt.begin() + classes, 0);
    for (int i = 0; i < n; i++)
      cnt[c[pn[i]]]++;
    for (int i = 1; i < classes; i++)
      cnt[i] += cnt[i - 1];
    for (int i = n - 1; i >= 0; i--)
      p[--cnt[c[pn[i]]]] = pn[i];

    cn[p[0]] = 0;
    classes = 1;
    for (int i = 1; i < n; i++) {
      pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
      pair<int, int> prev = {c[p[i - 1]], c[(p[i - 1] + (1 <<
h)) % n]};
      if (cur != prev)
        classes++;
      cn[p[i]] = classes - 1;
    }
    c.swap(cn);
  }

  return p;
}

vector<int> suffix_array(string s) {
  s += '$';
  vector<int> sorted_shifts = sort_cyclic_shifts(s);
  // remove sentinel position
  sorted_shifts.erase(sorted_shifts.begin());
  return sorted_shifts;
}

// --- Kasai LCP construction ---
vector<int> lcp_construction(const string &s, const vector<int>
&p) {
  int n = (int)s.size();
  vector<int> rank(n, 0);
  for (int i = 0; i < n; i++)
    rank[p[i]] = i;

  int k = 0;
```

```
  vector<int> lcp(max(0, n - 1), 0);
  for (int i = 0; i < n; i++) {
    if (rank[i] == n - 1) {
      k = 0;
      continue;
    }
    int j = p[rank[i] + 1];
    while (i + k < n && j + k < n && s[i + k] == s[j + k])
      k++;
    lcp[rank[i]] = k;
    if (k)
      k--;
  }
  return lcp;
}

// --- helpers: rank from SA and sparse table (RMQ) over lcp[]
---
vector<int> build_rank_from_sa(const vector<int> &p) {
  int n = (int)p.size();
  vector<int> rank(n, 0);
  for (int i = 0; i < n; ++i)
    rank[p[i]] = i;
  return rank;
}

vector<vector<int>> build_rmq(const vector<int> &lcp) {
  int m = (int)lcp.size();
  if (m == 0)
    return {};
  int LOG = 1;
  while ((1 << LOG) <= m)
    ++LOG;
  vector<vector<int>> st(LOG, vector<int>(m));
  st[0] = lcp;
  for (int k = 1; k < LOG; ++k) {
    int len = 1 << k;
    for (int i = 0; i + len <= m; ++i) {
      st[k][i] = min(st[k - 1][i], st[k - 1][i + (len >> 1)]);
    }
  }
  return st;
}

int rmq_query(const vector<vector<int>> &st, int l, int r) {
  if (st.empty())
    return 0;
  int len = r - l + 1;
  int k = 31 - __builtin_clz(len);
  return min(st[k][l], st[k][r - (1 << k) + 1]);
}

// --- LCP class ---
struct LCP {
  string s;
  int n = 0;
  vector<int> sa;         // suffix array
  vector<int> lcp;        // kasai lcp array (size n-1)
  vector<int> rank;       // rank[i] = position of suffix i in
sa
  vector<vector<int>> st; // sparse table on lcp[]

  // construct from a string (builds SA, LCP, rank and RMQ)
  LCP(const string &str = "") {
    if (!str.empty())
      build(str);
  }

  void build(const string &str) {
    s = str;
    n = (int)s.size();
    if (n == 0) {
```

```
      sa.clear();
      lcp.clear();
      rank.clear();
      st.clear();
      return;
    }
    sa = suffix_array(s);          // size n
    lcp = lcp_construction(s, sa); // size n-1 (or 0 if n==1)
    rank = build_rank_from_sa(sa);
    st = build_rmq(lcp);
  }

  // answer LCP of suffixes starting at i and j (0-indexed
positions in original
  // string)
  int query(int i, int j) const {
    if (i < 0 || j < 0 || i >= n || j >= n)
      return 0; // invalid -> 0
    if (i == j)
      return n - i; // full suffix matches itself
    int ri = rank[i], rj = rank[j];
    if (ri > rj)
      swap(ri, rj);
    // LCP of suffixes at sa[ri] and sa[rj] is min over
lcp[ri..rj-1]
    return rmq_query(st, ri, rj - 1);
  }

  // ---------------------------
  // Number of different substrings
  // ---------------------------
  // Explanation:
  // For a string of length n, total number of substrings
(including duplicates)
  // is:
  //   total = n * (n + 1) / 2
  //
  // When suffixes are sorted (by SA), each suffix p[i]
contributes
  //   (n - p[i]) - lcp[i-1]
  // new distinct substrings (prefixes that weren't counted
earlier).
  // Summing over i gives:
  //
  // distinct = sum_{i=0..n-1} (n - p[i]) - sum_{i=0..n-2}
lcp[i]
  // which simplifies to:
  // distinct = n*(n+1)/2 - sum_{i=0..n-2} lcp[i]
  //
  // We implement the compact formula below:
  long long distinct_substrings() const {
    if (n == 0)
      return 0;
    long long total = 1LL * n * (n + 1) / 2;
    long long sum_lcp = 0;
    for (int v : lcp)
      sum_lcp += v;
    return total - sum_lcp;
  }

  // optional accessors
  const vector<int> &get_sa() const { return sa; }
  const vector<int> &get_lcp_array() const { return lcp; }
};
```

5.8. manacher.h

```
vector<int> manacher_odd(string s) {
  int n = s.size();
  s = "$" + s + "^";
  vector<int> p(n + 2);
  int l = 0, r = 1;
  for (int i = 1; i <= n; i++) {
```

```
    p[i] = min(r - i, p[l + (r - i)]);
    while (s[i - p[i]] == s[i + p[i]]) {
      p[i]++;
    }
    if (i + p[i] > r) {
      l = i - p[i];
      r = i + p[i];
    }
  }
  return vector<int>(begin(p) + 1, end(p) - 1);
}

// gives a vector of size 2*n - 1 each element of which gives
the length of
// longest palindrome centred at ith location
// if i is even(0,2,4 .. etc.) center is an element -> odd
palindromes
// and if i is odd(1, 3, 5 .. etc.) center is between two
elements -> even
// palindromes
vector<int> manacher(string s) {
  string t;
  for (auto c : s) {
    t += string("#") + c;
  }
  auto res = manacher_odd(t + "#");
  for (auto &e : res) {
    e = e - 1;
  }
  return vector<int>(begin(res) + 1, end(res) - 1);
}
```

5.9. prefix_func.h

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

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

5.10. suffix_array.h

```
vector<int> sort_cyclic_shifts(const string &a) {
  int n = a.size();
  const int alphabet = 256;

  vector<int> p(n), c(n);
  vector<int> cnt(max(alphabet, n), 0);

  // initial sort for h = 0
  for (int i = 0; i < n; i++)
    cnt[(unsigned char)a[i]]++;
```

```
  for (int i = 1; i < alphabet; i++)
    cnt[i] = cnt[i - 1] + cnt[i];
  for (int i = n - 1; i >= 0; i--)
    p[--cnt[(unsigned char)a[i]]] = i;

  c[p[0]] = 0;
  int classes = 1;
  for (int i = 1; i < n; i++) {
    if (a[p[i]] != a[p[i - 1]])
      classes++;
    c[p[i]] = classes - 1;
  }

  for (int h = 0; (1 << h) < n; h++) {
    vector<int> pn(n), cn(n);

    // first we are sorting on the basis of second half ...
    for (int i = 0; i < n; i++) {
      pn[i] = p[i] - (1 << h);
      if (pn[i] < 0)
        pn[i] += n;
    }

    // now we sort on the base of first half ... (stable
counting sort)
    fill(cnt.begin(), cnt.begin() + classes, 0);
    for (int i = 0; i < n; i++)
      cnt[c[pn[i]]]++;
    for (int i = 1; i < classes; i++)
      cnt[i] += cnt[i - 1];
    for (int i = n - 1; i >= 0; i--)
      p[--cnt[c[pn[i]]]] = pn[i];

    // now we evaluate new classes
    cn[p[0]] = 0;
    classes = 1;
    for (int i = 1; i < n; i++) {
      pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
      pair<int, int> prev = {c[p[i - 1]], c[(p[i - 1] + (1 <<
h)) % n]};
      if (cur != prev)
        classes++;
      cn[p[i]] = classes - 1;
    }
    c.swap(cn);
  }

  return p;
}

vector<int> suffix_array(string s) {
  s += '$';
  vector<int> sorted_shifts = sort_cyclic_shifts(s);
  // remove sentinel position
  sorted_shifts.erase(sorted_shifts.begin());
  return sorted_shifts;
}
```

5.11. z_func.h

```
vector<int> z_function(const string &s) {
  int n = (int)s.size();
  if (n == 0)
    return {};
  vector<int> z(n, 0);
  int l = 0, r = 0;
  for (int i = 1; i < n; ++i) {
    if (i <= r)
      z[i] = 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;
}

vector<int> z_function_int(const vector<long long> &v) {
  int n = (int)v.size();
  if (n == 0)
    return {};
  vector<int> z(n, 0);
  int l = 0, r = 0;
  for (int i = 1; i < n; ++i) {
    if (i <= r)
      z[i] = min(r - i + 1, z[i - l]);
    while (i + z[i] < n && v[z[i]] == v[i + z[i]])
      ++z[i];
    if (i + z[i] - 1 > r) {
      l = i;
      r = i + z[i] - 1;
    }
  }
  return z;
}
```

6. DP

6.1. edit_distance.h

```
int edit_distance(string a, string b) {
  int n = a.size(), m = b.size();
  vector<int> dp(m + 1), ndp(m + 1);
  iota(dp.begin(), dp.end(), 0);
  for (int i = 1; i <= n; i++) {
    ndp[0] = i;
    for (int j = 1; j <= m; j++)
      ndp[j] = a[i - 1] == b[j - 1] ? dp[j - 1]
                                    : 1 + min({dp[j - 1],
dp[j], ndp[j - 1]});
    swap(dp, ndp);
  }
  return dp[m];
}
```

6.2. lcs.h

```
int lcs(string a, string b) {
  int n = a.size(), m = b.size();
  vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
      dp[i][j] = a[i - 1] == b[j - 1] ? 1 + dp[i - 1][j - 1]
                                      : max(dp[i - 1][j], dp[i]
[j - 1]);
  return dp[n][m];
}
```

6.3. sos_dp.h

```
// SOS DP: f[mask] = sum of f[submask] for all submask ⊆ mask
void sos_dp(vector<long long> &f, int n) {
  for (int i = 0; i < n; i++)
    for (int mask = 0; mask < (1 << n); mask++)
      if (mask & (1 << i))
        f[mask] += f[mask ^ (1 << i)];
}

// SOSU DP: f[mask] = sum of f[supmask] for all supmask ⊇ mask
void sosu_dp(vector<long long> &f, int n) {
  for (int i = 0; i < n; i++)
    for (int mask = 0; mask < (1 << n); mask++)
      if (!(mask & (1 << i)))
        f[mask] += f[mask | (1 << i)];
}
```

6.4. subset_sum.h

```
int subset_sum(vector<int> &a, int sum) {
  vector<int> dp(sum + 1, 0);
  dp[0] = 1;
  for (int x : a)
    for (int s = sum; s >= x; s--)
      dp[s] += dp[s - x];
  return dp[sum];
}
```

7. Geometry

7.1. all.h

```
#ifndef GEOMETRY_H
#define GEOMETRY_H

#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

using namespace std;

typedef long double ld;
const ld EPS = 1e-9;
const ld PI = acos(-1.0);

inline int sign(ld x) { return (x > EPS) - (x < -EPS); }
inline ld sq(ld x) { return x * x; }

struct P {
  ld x, y;
  P(ld x = 0, ld y = 0) : x(x), y(y) {}
  P operator+(const P &o) const { return P(x + o.x, y + o.y); }
  P operator-(const P &o) const { return P(x - o.x, y - o.y); }
  P operator*(ld s) const { return P(x * s, y * s); }
  P operator/(ld s) const { return P(x / s, y / s); }
  ld dot(const P &o) const { return x * o.x + y * o.y; }
  ld cross(const P &o) const { return x * o.y - y * o.x; }
  ld dist2() const { return x * x + y * y; }
  ld dist() const { return sqrt(dist2()); }
  P rotate(ld ang) const {
    return P(x * cos(ang) - y * sin(ang), x * sin(ang) + y *
cos(ang));
  }
  P perp() const { return P(-y, x); }
  P unit() const { return *this / dist(); }
  bool operator<(const P &o) const {
    return sign(x - o.x) != 0 ? x < o.x : sign(y - o.y) < 0;
  }
  bool operator==(const P &o) const {
    return sign(x - o.x) == 0 && sign(y - o.y) == 0;
  }
  friend ostream &operator<<(ostream &os, const P &p) {
    return os << "(" << p.x << ", " << p.y << ")";
  }
};

ld cross(P a, P b, P c) { return (b - a).cross(c - a); }

struct Line {
  ld a, b, c; // ax + by = c
  Line(ld a, ld b, ld c) : a(a), b(b), c(c) {}
  Line(P p1, P p2) : a(p1.y - p2.y), b(p2.x - p1.x), c(a * p1.x
+ b * p1.y) {}
  ld eval(P p) const { return a * p.x + b * p.y - c; }
  ld norm2() const { return a * a + b * b; }
  bool parallel(const Line &o) const { return sign(a * o.b -
o.a * b) == 0; }
  bool intersect(const Line &o, P &res) const {
    ld det = a * o.b - o.a * b;
    if (sign(det) == 0)
```

```
      return false;
    res.x = (c * o.b - o.c * b) / det;
    res.y = (a * o.c - o.a * c) / det;
    return true;
  }
  ld dist(P p) const { return abs(eval(p)) / sqrt(norm2()); }
  P proj(P p) const {
    ld d = eval(p) / norm2();
    return P(p.x - a * d, p.y - b * d);
  }
  P refl(P p) const { return proj(p) * 2 - p; }
};

struct Seg {
  P s, e;
  Seg(P s, P e) : s(s), e(e) {}
  Line to_line() const { return Line(s, e); }
  bool on_seg(P p) const {
    return sign(cross(s, e, p)) == 0 && sign((p - s).dot(p -
e)) <= 0;
  }
  ld dist(P p) const {
    if ((p - s).dot(e - s) < 0)
      return (p - s).dist();
    if ((p - e).dot(s - e) < 0)
      return (p - e).dist();
    return to_line().dist(p);
  }
};

ld area(const vector<P> &p) {
  ld a = 0;
  for (int i = 0; i < p.size(); i++)
    a += p[i].cross(p[(i + 1) % p.size()]);
  return a / 2.0;
}

bool is_convex(const vector<P> &p) {
  bool has_pos = false, has_neg = false;
  for (int i = 0; i < p.size(); i++) {
    int o = sign(cross(p[i], p[(i + 1) % p.size()], p[(i + 2) %
p.size()]));
    if (o > 0)
      has_pos = true;
    if (o < 0)
      has_neg = true;
  }
  return !(has_pos && has_neg);
}

int in_poly(const vector<P> &p, P pt) { // 1: in, -1: on, 0:
out
  bool in = false;
  for (int i = 0; i < p.size(); i++) {
    P a = p[i] - pt, b = p[(i + 1) % p.size()] - pt;
    if (a.y > b.y)
      swap(a, b);
    if (sign(a.cross(b)) == 0 && sign(a.dot(b)) <= 0)
      return -1;
    if (a.y <= 0 && b.y > 0 && sign(a.cross(b)) > 0)
      in = !in;
  }
  return in;
}

vector<P> convex_hull(vector<P> pts) {
  if (pts.size() <= 2)
    return pts;
  sort(pts.begin(), pts.end());
  vector<P> h;
  for (int i = 0; i < 2; i++) {
    int start = h.size();
```

```
    for (auto &pt : pts) {
      while (h.size() >= start + 2 && cross(h[h.size() - 2],
h.back(), pt) <= 0)
        h.pop_back();
      h.push_back(pt);
    }
    h.pop_back();
    reverse(pts.begin(), pts.end());
  }
  return h;
}

struct Circle {
  P c;
  ld r;
  bool in(P p) { return sign((p - c).dist() - r) <= 0; }
  int intersect(Line l, vector<P> &res) {
    ld d = l.dist(c);
    if (sign(d - r) > 0)
      return 0;
    P p = l.proj(c);
    if (sign(d - r) == 0) {
      res.push_back(p);
      return 1;
    }
    P dir = P(l.b, -l.a).unit();
    ld h = sqrt(max((ld)0.0, r * r - d * d));
    res.push_back(p + dir * h);
    res.push_back(p - dir * h);
    return 2;
  }
};

#endif
```

8. Math

8.1. fft.h

```
#ifndef FFT_H
#define FFT_H

#include <algorithm>
#include <cmath>
#include <complex>
#include <vector>

typedef std::complex<double> cd;
const double PI = std::acos(-1);

void fft(std::vector<cd> &a, bool invert) {
  int n = a.size();
  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1)
      j ^= bit;
    j ^= bit;
    if (i < j)
      std::swap(a[i], a[j]);
  }
  for (int len = 2; len <= n; len <<= 1) {
    double ang = 2 * PI / len * (invert ? -1 : 1);
    cd wlen(std::cos(ang), std::sin(ang));
    for (int i = 0; i < n; i += len) {
      cd w(1);
      for (int j = 0; j < len / 2; j++) {
        cd u = a[i + j], v = a[i + j + len / 2] * w;
        a[i + j] = u + v;
        a[i + j + len / 2] = u - v;
        w *= wlen;
      }
    }
```

```
  }
  if (invert)
    for (cd &x : a)
      x /= n;
}

std::vector<long long> poly_multiply(const std::vector<long
long> &a,
                                     const std::vector<long
long> &b) {
  std::vector<cd> fa(a.begin(), a.end()), fb(b.begin(),
b.end());
  int n = 1;
  while (n < a.size() + b.size())
    n <<= 1;
  fa.resize(n);
  fb.resize(n);

  fft(fa, false);
  fft(fb, false);
  for (int i = 0; i < n; i++)
    fa[i] *= fb[i];
  fft(fa, true);

  std::vector<long long> result(n);
  for (int i = 0; i < n; i++)
    result[i] = (long long)std::round(fa[i].real());
  while (result.size() > 1 && result.back() == 0)
    result.pop_back();
  return result;
}

#endif
```

8.2. gauss.h

```
#ifndef ICPC_SLAE_H
#define ICPC_SLAE_H

#include <algorithm>
#include <climits>
#include <cmath>
#include <vector>

namespace cp_algo {

using Real = double;
constexpr Real EPS = 1e-9;
constexpr int INF_SOLS = INT_MAX;

int solve_slae(std::vector<std::vector<Real>> &a,
std::vector<Real> &ans) {
  if (a.empty())
    return 0;
  int n = (int)a.size();
  int m = (int)a[0].size() - 1;

  std::vector<int> where(m, -1);
  int row = 0;
  for (int col = 0; col < m && row < n; ++col) {
    int sel = row;
    for (int i = row; i < n; ++i) {
      if (std::abs(a[i][col]) > std::abs(a[sel][col]))
        sel = i;
    }

    if (std::abs(a[sel][col]) < EPS)
      continue;

    std::swap(a[sel], a[row]);
    where[col] = row;
```

```
    Real div = a[row][col];
    for (int j = col; j <= m; ++j)
      a[row][j] /= div;

    for (int i = 0; i < n; ++i) {
      if (i != row) {
        Real mult = a[i][col];
        for (int j = col; j <= m; ++j)
          a[i][j] -= mult * a[row][j];
      }
    }
    ++row;
  }

  ans.assign(m, 0);
  for (int j = 0; j < m; ++j) {
    if (where[j] != -1)
      ans[j] = a[where[j]][m];
  }

  for (int i = 0; i < n; ++i) {
    Real sum = 0;
    for (int j = 0; j < m; ++j)
      sum += ans[j] * a[i][j];
    if (std::abs(sum - a[i][m]) > EPS)
      return 0;
  }

  for (int j = 0; j < m; ++j) {
    if (where[j] == -1)
      return INF_SOLS;
  }

  return 1;
}
```

```
} // namespace cp_algo

#endif
```

8.3. ntt.h

```
#ifndef NTT_H
#define NTT_H

#include <algorithm> // For std::swap
#include <iostream>
#include <vector>

namespace NTT {

using ll = long long;

const ll MOD = 998244353;
const ll ROOT = 3;

ll power(ll a, ll p) {
  ll res = 1;
  a %= MOD;
  while (p > 0) {
    if (p & 1)
      res = (res * a) % MOD;
    a = (a * a) % MOD;
    p >>= 1;
  }
  return res;
}

ll modInverse(ll a) { return power(a, MOD - 2); }

void ntt(std::vector<ll> &a, bool invert) {
  int n = a.size();
  if (n == 1)
```

```
    return;

  for (int i = 1, j = 0; i < n; i++) {
    int bit = n >> 1;
    for (; j & bit; bit >>= 1) {
      j ^= bit;
    }
    j ^= bit;
    if (i < j) {
      std::swap(a[i], a[j]);
    }
  }

  for (int len = 2; len <= n; len <<= 1) {
    ll wlen = power(ROOT, (MOD - 1) / len);
    if (invert) {
      wlen = modInverse(wlen);
    }
    for (int i = 0; i < n; i += len) {
      ll w = 1;
      for (int j = 0; j < len / 2; j++) {
        ll u = a[i + j];
        ll v = (a[i + j + len / 2] * w) % MOD;
        a[i + j] = (u + v) % MOD;
        a[i + j + len / 2] = (u - v + MOD) % MOD;
        w = (w * wlen) % MOD;
      }
    }
  }

  if (invert) {
    ll n_inv = modInverse(n);
    for (ll &x : a) {
      x = (x * n_inv) % MOD;
    }
  }
}

std::vector<ll> poly_multiply_ntt(std::vector<ll> a,
std::vector<ll> b) {
  int res_size = a.size() + b.size() - 1;
  int n = 1;
  while (n < res_size) {
    n <<= 1;
  }

  a.resize(n);
  b.resize(n);

  ntt(a, false);
  ntt(b, false);

  for (int i = 0; i < n; i++) {
    a[i] = (a[i] * b[i]) % MOD;
  }

  ntt(a, true);

  a.resize(res_size); // Truncate to the correct polynomial
size
  return a;
}

} // namespace NTT

#endif // NTT_H
```

9. Misc

9.1. countinversions.h

```
ll count_inversion(vector<ll> &a) {
```

```
  int n = a.size();
  vector<ll> tmp(n);
  function<ll(int, int)> ms = [&](int l, int r) -> ll {
    if (r - l <= 1)
      return 0;
    int m = (l + r) / 2;
    ll inv = ms(l, m) + ms(m, r);
    int i = l, j = m, k = l;
    while (i < m || j < r) {
      if (j == r || (i < m && a[i] <= a[j]))
        tmp[k++] = a[i++];
      else {
        tmp[k++] = a[j++];
        inv += m - i;
      }
    }
    for (int t = l; t < r; ++t)
      a[t] = tmp[t];
    return inv;
  };
  return ms(0, n);
}
```

9.2. lis.h

```
int LIS(vector<int> &a) {
  vector<int> v;
  for (int x : a) {
    auto it = lower_bound(v.begin(), v.end(), x);
    if (it == v.end())
      v.push_back(x);
    else
      *it = x;
  }
  return (int)v.size();
}
```

10. Tricks

10.1. geom.h

```
Basic Utilities
EPS-based floating point comparisons (sign).
Square function (sq).

2D Points / Vectors (struct P)
Vector arithmetic (+, -, *, /).
Dot and Cross products.
Distance calculations (dist, dist2).
Rotation (rotate) and Perpendicular vector (perp).
Unit vector normalization (unit).
Comparators for sorting (<, ==) and output stream support (<<).

Lines (struct Line)Defined as $ax + by = c$.
Parallel check.
Intersection with another line.
Distance from a point to the line.
Orthogonal projection of a point onto the line.
Reflection of a point across the line.

Segments (struct Seg)
Check if a point is strictly on the segment.
Shortest distance from a point to the segment (clamped to
endpoints).
PolygonsSigned area.
Convexity check.
Point in polygon test (returns: inside, outside, or on
boundary).
Convex Hull (Monotone Chain algorithm).

Circles (struct Circle)
Check if a point is inside or on the boundary.
Intersection with a line (finds 0, 1, or 2 points)
```

10.2. trees.h

```
1. Given a bridge tree,
    the minimum number of edges required to be added to make
the graph 2 -
        edge
            connected = ceil(L / 2) where L are the number of
leaves
                            .2. Block cut tree builds a
bipartite
                            graph(original node->block node
                                      ->original node(may be
articulation point)
                                      ->...)
                            .Non -
                        articulation points have degree = 1,
            and articulation points have degree >=
                2(connecting two or more graphs).
```