Binary Trie

\\ Base code //

```cpp
template<int Log = 62> // control numbe of bits
class binary_trie {
    struct node {
        int cnt{};
        node* mp[2]{};
    } *root = new node;

    void clear(node* x) {
        if (!x) return;
        for (auto& i : x->mp) clear(i);
        delete x;
    }

public:
//    ~trie_xor() { clear(root); }


    void clear() {
        clear(root);
        root = new node;
    }
};
```

\\ Erase code //

```cpp
// Erase number `n` from the trie
void erase(int n) {
    node *cur = root;
    for (int i = Log; i >= 0; i--) {
        bool idx = (n >> i) & 1LL;
        node *next = cur->mp[idx];
        next->cnt--;
        if (next->cnt == 0) {
            delete next;
            cur->mp[idx] = nullptr;
            return;
        }
        cur = next;
    }
}
```

\\ Add code //

```cpp
// Insert number `num` with count `c` (default = 1)
void add(int num, int c = 1) {
    node* x = root;
    for (int i = Log; i >= 0; --i) {
        x->cnt += c;
        bool b = (num >> i) & 1LL;
        if (!x->mp[b]) x->mp[b] = new node;
        x = x->mp[b];
    }
    x->cnt += c;
}
```

\\ Contains code //

```cpp
// Return whether number exists in the trie
bool contains(int num) {
    node* x = root;
    for (int i = Log; i >= 0; --i) {
        bool b = (num >> i) & 1LL;
        if (!x->mp[b] || x->mp[b]->cnt == 0) return false;
        x = x->mp[b];
    }
    return x->cnt > 0;
}
```

Trie String

\\ Min XOR code //

```cpp
// Return the number in the trie that gives min xor with `num`
int min_xor(int num) {
    if (root->cnt == 0) return -1;
    node* x = root;
    int ans = 0;
    for (int i = Log; i >= 0; --i) {
        bool b = (num >> i) & 1LL;
        if (x->mp[b]) {
            x = x->mp[b];
        } else {
            ans |= (1LL << i);
            x = x->mp[(!b)];
        }
    }
    return ans;
}
```

\\ Max XOR code //

```cpp
// Return the number in the trie that gives max xor with `num`
int max_xor(int num) {
    if (root->cnt == 0) return -1;
    node* x = root;
    int ans = 0;
    for (int i = Log; i >= 0; --i) {
        bool b = ((num >> i) & 1LL) ^ 1LL;
        if (x->mp[b]) {
            ans |= (1LL << i);
            x = x->mp[b];
        } else {
            x = x->mp[(!b)];
        }
    }
    return ans;
}
```

\\ Base code //

```cpp
class TrieString {
    struct Node {
        Node* child[26]{};
        // unordered_map<char, Node*> child; // Uncomment this line to use unordered_map for flexibility

        int wordCount = 0;
        int prefixCount = 0;
        char value = 'a';

        Node() = default;
    };

    Node* root;

public:
    TrieString() : root(new Node()) {}


    void clear(Node* node) {
        if (!node) return;
        for (Node* child : node->child) clear(child);
        delete node;
    }

//    ~TrieString() clear(root);

};
```

\\ Insert code //

```cpp
void insert(const string& s) {
    Node* cur = root;
    for (char ch : s) {
        int idx = ch - cur->value;
        if (!cur->child[idx]) cur->child[idx] = new Node();
        cur = cur->child[idx];
        cur->prefixCount++;
    }
    cur->wordCount++;
}
```

\\ Contains code //

```
bool contains(const string& s) const {
    Node* cur = root;
    for (char ch : s) {
        int idx = ch - cur->value;
        if (!cur->child[idx]) return false;
        cur = cur->child[idx];
    }
    return cur->wordCount > 0;
}
```

\\ Count words code //

```
int countWord(const string& s) const {
    Node* cur = root;
    for (char ch : s) {
        int idx = ch - cur->value;
        if (!cur->child[idx]) return 0;
        cur = cur->child[idx];
    }
    return cur->wordCount;
}
```

\\ Count prefix code //

```
int countPrefix(const string& s) const {
    Node* cur = root;
    for (char ch : s) {
        int idx = ch - cur->value;
        if (!cur->child[idx]) return 0;
        cur = cur->child[idx];
    }
    return cur->prefixCount;
}
```

\\ Erase code //

```
void erase(const string& s) {
    Node* cur = root;

    for (char ch : s) {
        int idx = ch - cur->value;
        if (!cur->child[idx]) return;

        Node *next = cur->child[idx];
        next->prefixCount--;

        if(next->prefixCount == 0){
            delete next;
            cur->child[idx] = nullptr;
            return;
        }

        cur = next;
    }

    if (cur->wordCount == 0) return; // word not present
    cur->wordCount--;
}
```

Sum Of All

```
ll sumOfAll(ll x)
{
    return ((x + (x % 2)) / 2) * (x + (x % 2 == 0));
}
```

Sparse Table

\\ All The code //

```cpp
template<typename T>
struct sparse{
    int Log, n;
    vector<vector<T>> table;
    function<T(T, T)> merge;
    template<class U>
    explicit sparse(vector<T> arr, U merge) : merge(merge), n((int)arr.size()), Log(__lg(arr.size()) + 1), table(Log, vector<T>(n)) {
        table[0] = arr;
        for(int l = 1; l < Log; l++) {
            for(int i = 0; i + (1 << (l - 1)) < n; i++) {
                table[l][i] = merge(table[l - 1][i], table[l - 1][i + (1 << (l - 1))]);
            }
        }
    }
    T query(int l, int r) {
        if(l > r) return {};
        int len = __lg(r - l + 1);
        return merge(table[len][l], table[len][r - (1 << len) + 1]);
    }
};
```

\\ How to use //

```cpp
vector<int> arr = {1, 3, 2, 5, 4};
auto merge_operation = [](int a, int b) { return max(a,  b); };
sparse<int> sparseTable(arr, merge_operation);
```

BIT

```cpp
template<class T>
struct BIT { // 0-based
    int n;
    vector<T> tree;
    explicit BIT(int size) : n(size + 5), tree(n + 1) { }

    void add(int i, T val) {
        for (i++; i <= n; i += i & -i)
            tree[i] += val;
    }

    T query(int i) {
        T sum = 0;
        for (i++; i > 0; i -= i & -i)
            sum += tree[i];
        return sum;
    }

    T range(int l, int r) {
        if(l > r) return T();
        return query(r) - query(l - 1);
    }

    int lower_bound(T target) {
        if(target <= 0) return 0;
        int pos = 0;
        T sum = 0;
        for (int i = 1 << __lg(n); i > 0; i >>= 1) {
            if(pos + i <= n && sum + tree[pos + i] < target) {
                sum += tree[pos + i];
                pos += i;
            }
        }
        return pos;
    }
};
```

BIT Range

```cpp
template<typename T>
class BITRange { // 0-based
    int n;
    vector<T> B1, B2;

    void add(vector<T>& bit, int i, T x) {
        for (++i; i <= n; i += i & -i)
            bit[i] += x;
    }

    T query(vector<T>& bit, int i) {
        T res = 0;
        for (++i; i > 0; i -= i & -i)
            res += bit[i];
        return res;
    }

public:
    explicit BITRange(int size) : n(size + 5), B1(n + 2), B2(n + 2) {}

    void add(int l, int r, T x) {
        add(B1, l, x);
        add(B1, r + 1, -x);
        add(B2, l, x * (l - 1));
        add(B2, r + 1, -x * r);
    }
    void add(int i, T x) { add(B2, i, -x); }

    T query(int i) {
        return query(B1, i) * i - query(B2, i);
    }

    T range(int l, int r) {
        if (l > r) return T();
        return query(r) - query(l - 1);
    }
};
```

SQRT & MO

```cpp
struct query{
    int l, r, idx;

    query(){};
};


void solve(){
    int n, q, k;
    cin >> n >> q >> k;
    getVec(num, n);

    int Z = sqrt(n) + 1;

    vector<int> ans(q);
    vector<query> qu(q);

    for(int i=0, l, r; i<q; i++){
        cin >> l >> r;
        l--, r--;
        qu[i].l = l, qu[i].r = r, qu[i].idx = i;
    }

    sort(BegEnd(qu), [&](query &a, query &b){
        if(a.l / Z == b.l / Z){
            return a.r < b.r;
        }
        else{
            return a.l < b.l;
        }
    });

    ll tmp = 0;
    auto add = [&](int idx){
        // code
    };
    auto rem = [&](int idx){
        // code
    };

    int l = 0, r = -1;
    for(auto &x: qu){
        while(l > x.l){
            l--;
            add(l);
        }
        while(r < x.r){
            r++;
            add(r);
        }
        while(r > x.r){
            rem(r);
            r--;
        }
        while(l < x.l){
            rem(l);
            l++;
        }
        ans[x.idx] = tmp;
    }

    for(auto &x: ans) cout << x << endl;
}
```