针对色板游戏这道题，我们依次介绍三种不同效率的算法：**暴力模拟**、**分块法**、**线段树 + 状态压缩**。每种算法均会给出思想、正确性证明以及带详细注释的完整 C++ 代码。

算法一：暴力模拟

1. 算法思想与步骤

- 用一个数组 `color[1..L]` 直接存储每个方格当前的颜色，初始全为 1。
- 对于涂色操作 `C A B C`：
  - 若 $A > B$，交换 A, B；
  - 遍历 $i \in [A, B]$，执行 `color[i] = C`。
- 对于查询操作 `P A B`：
  - 若 $A > B$，交换 A, B；
  - 用一个 `bool` 数组 `vis[1..T]` 标记出现过的颜色；
  - 遍历 $i \in [A, B]$，将 `vis[color[i]]` 置为 `true`；
  - 统计并输出 `vis` 中为真的个数。

2. 正确性证明

数组直接保存了每个位置的颜色，修改和查询均严格依照题目要求模拟，正确性显然。

3. 时间复杂度

- 每次操作均扫描整个区间，最坏代价 $\Theta(L)$。
- 总复杂度 $O(O \cdot L)$。在 $L, O \le 10^5$ 时会超时，仅用于小规模验证。

4. C++ 代码

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

int main() {
    freopen("color.in", "r", stdin);
    freopen("color.out", "w", stdout);

    int L, T, O;
    cin >> L >> T >> O;

    // 下标从1开始，初始全为颜色1
    vector<int> color(L + 1, 1);

    while (O--) {
        char op;
        cin >> op;
        if (op == 'C') {
            int A, B, C;
            cin >> A >> B >> C;
            if (A > B) swap(A, B);          // 保证 A <= B
            // 暴力涂色
            for (int i = A; i <= B; ++i) {
                color[i] = C;
            }
        } else { // op == 'P'
            int A, B;
            cin >> A >> B;
            if (A > B) swap(A, B);

            // 统计区间内出现的颜色种类
            vector<bool> vis(T + 1, false);
            for (int i = A; i <= B; ++i) {
                vis[color[i]] = true;
            }
            int ans = 0;
            for (int c = 1; c <= T; ++c) {
                if (vis[c]) ++ans;
```

```
            }
            cout << ans << "\n";
        }
    }

    return 0;
}
```

算法二：分块法

1. 算法思想与步骤

将 L 个格子划分为大小为 $B \approx \sqrt{L}$ 的块，共约 $\lceil L/B \rceil$ 块。每个块维护：

- `lazy`：若整块被统一颜色覆盖，则记录该颜色（值为 -1 表示无统一颜色）；
- `mask`：该块当前包含的颜色集合（用 T 位二进制表示，第 k 位为 1 表示存在颜色 $k+1$）。

操作处理：

- **区间涂色** `C A B C`：
  1. 对于被区间完全覆盖的**整块**，直接设置 `lazy = C`，`mask = 1 << (C-1)`。
  2. 对于两端的**零散部分**，先执行 `push_down`（若该块有懒标记，则把整块刷成懒标记颜色），然后暴力修改数组对应位置，再重新计算该块的 `mask`。
- **区间查询** `P A B`：
  1. 对于被区间完全覆盖的整块，直接将其 `mask` 并入答案。
  2. 对于零散部分，先 `push_down`，再暴力检查每个位置的颜色，手工加入答案集合。
  3. 答案集合的二进制表示中 1 的个数即为颜色种数。

2. 正确性证明

- `lazy` 标记的含义为“整个块都已是该颜色”。当块被完全覆盖时，块内原有颜色信息无需保留，直接将 `mask` 设为该颜色对应位即可。
- 访问零散部分时，必须先下推懒标记，保证数组值与块信息一致。
- 查询时，完整块直接使用 `mask`（即该块颜色的并集），零散部分在同步后暴力统计，最终合并得到的集合恰好是区间出现的所有颜色。

3. 时间复杂度

设块大小为 B，则块数约为 L/B。每次操作处理的整块数 $O(L/B)$，零散元素数 $O(B)$。取 $B = \sqrt{L} \approx 316$，则单次操作复杂度为 $O(\sqrt{L})$。总复杂度 $O(O\sqrt{L})$，约 3×10^7，可在时限内通过。

4. C++ 代码

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

const int MAXL = 100005;
const int MAXB = 320;                // sqrt(100000) ≈ 316

int L, T, O;
int color[MAXL];                     // 实际颜色值（仅在块被部分修改时维护）
int blockId[MAXL];                   // 每个位置属于哪个块
int blockL[MAXB], blockR[MAXB];      // 每个块的左右边界（1-based）
int lazy[MAXB];                      // 懒标记，-1表示无统一颜色，否则为统一颜
色
int mask[MAXB];                      // 块内颜色集合的位掩码
int blockCount;                      // 总块数

// 将块的懒标记下推到数组，并清空懒标记
void pushDown(int bid) {
    if (lazy[bid] != -1) {
        int c = lazy[bid];
        for (int i = blockL[bid]; i <= blockR[bid]; ++i) {
            color[i] = c;
        }
        lazy[bid] = -1;
    }
}

// 重新计算块的颜色掩码（必须在块无懒标记时调用）
void rebuildMask(int bid) {
    int m = 0;
    for (int i = blockL[bid]; i <= blockR[bid]; ++i) {
        m |= (1 << (color[i] - 1));
    }
    mask[bid] = m;
}

// 区间涂色
void update(int l, int r, int c) {
```

```
    if (l > r) swap(l, r);
    int lid = blockId[l], rid = blockId[r];

    if (lid == rid) {
        // 同一块内暴力修改
        pushDown(lid);
        for (int i = l; i <= r; ++i) color[i] = c;
        rebuildMask(lid);
    } else {
        // 左端零散块
        pushDown(lid);
        for (int i = l; i <= blockR[lid]; ++i) color[i] = c;
        rebuildMask(lid);
        // 中间整块
        for (int bid = lid + 1; bid < rid; ++bid) {
            lazy[bid] = c;
            mask[bid] = 1 << (c - 1);    // 整块只有一种颜色
        }
        // 右端零散块
        pushDown(rid);
        for (int i = blockL[rid]; i <= r; ++i) color[i] = c;
        rebuildMask(rid);
    }
}

// 区间查询
int query(int l, int r) {
    if (l > r) swap(l, r);
    int lid = blockId[l], rid = blockId[r];
    int ansMask = 0;

    if (lid == rid) {
        pushDown(lid);
        for (int i = l; i <= r; ++i) {
            ansMask |= (1 << (color[i] - 1));
        }
    } else {
```

```
        // 左端零散
        pushDown(lid);
        for (int i = l; i <= blockR[lid]; ++i) {
            ansMask |= (1 << (color[i] - 1));
        }
        // 中间整块直接取mask
        for (int bid = lid + 1; bid < rid; ++bid) {
            ansMask |= mask[bid];
        }
        // 右端零散
        pushDown(rid);
        for (int i = blockL[rid]; i <= r; ++i) {
            ansMask |= (1 << (color[i] - 1));
        }
    }
    // 统计位掩码中1的个数
    return __builtin_popcount(ansMask);
}

int main() {
    freopen("color.in", "r", stdin);
    freopen("color.out", "w", stdout);

    cin >> L >> T >> O;
    // 初始全为颜色1
    for (int i = 1; i <= L; ++i) color[i] = 1;

    // 分块，块大小约 sqrt(L)
    int B = max(1, (int)sqrt(L));
    blockCount = 0;
    for (int i = 1; i <= L; i += B) {
        ++blockCount;
        blockL[blockCount] = i;
        blockR[blockCount] = min(i + B - 1, L);
        lazy[blockCount] = -1; // 无统一颜色
        for (int j = blockL[blockCount]; j <= blockR[blockCount]; ++j) {
            blockId[j] = blockCount;
```

```
        }
    }
    // 初始化所有块的掩码（全1颜色）
    for (int bid = 1; bid <= blockCount; ++bid) {
        rebuildMask(bid);
    }

    while (O--) {
        char op;
        cin >> op;
        if (op == 'C') {
            int A, B, C;
            cin >> A >> B >> C;
            update(A, B, C);
        } else {
            int A, B;
            cin >> A >> B;
            cout << query(A, B) << "\n";
        }
    }

    return 0;
}
```

算法三：线段树 + 状态压缩

1. 算法思想与步骤

线段树每个节点表示一个区间，维护两个信息：

- `cols`：该区间内出现的颜色集合，用 T 位二进制表示（$T \le 30$，可使用 32 位整数）；
- `lazy`：懒标记，0 表示无标记，非 0 表示整个区间被染成该颜色。

操作：

- **建树**：初始区间全为颜色 1，故所有节点的 `cols` 均为 $2^0 = 1$，`lazy = 0`。
- **区间涂色** `update(l, r, c)`：

1. 若当前节点区间完全在 $[l, r]$ 内，则打上懒标记 `lazy = c`，并将 `cols` 置为 $1 \ll (c-1)$，返回。
2. 否则先 `pushDown`（将懒标记下传给左右儿子），然后递归更新左右儿子，最后用左右儿子的 `cols` 取按位或合并。

- **区间查询** `query(l, r)`：
  1. 若完全覆盖，返回当前节点的 `cols`。
  2. `pushDown` 后递归查询左右儿子，将结果按位或合并返回。
- 最后，查询结果的二进制中 1 的个数即为答案（使用 `__builtin_popcount`）。

2. 正确性证明

- 位掩码的第 k 位代表颜色 $k+1$ 是否存在。`按位或` 运算完美对应集合的并操作，因此向上合并 `cols` 能正确得到区间颜色集合。
- 懒标记表示该区间内所有位置已统一为颜色 c，此时 `cols` 仅包含该颜色，下传时直接将左右儿子设为同一状态，保证了信息一致性。
- 线段树的递归结构保证每个修改/查询操作都能在 $O(\log L)$ 时间内被分解为不重叠的区间，正确性由线段树的基本性质保证。

3. 时间复杂度

单次操作仅涉及线段树的 $O(\log L)$ 个节点，每个节点操作为 $O(1)$ 的位运算。总复杂度 $O(O \log L)$，在 $L, O \le 10^5$ 时约为 1.7×10^6 次操作，非常高效。

4. C++ 代码

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

const int MAXL = 100005;

int L, T, O;

// 线段树节点：存储颜色掩码 cols 和懒标记 lazy
struct Node {
    int cols;   // 区间内颜色的位掩码，第 i 位为1代表存在颜色 i+1
    int lazy;   // 懒标记，0表示无标记，否则表示区间被染成该颜色
} tree[MAXL * 4];

// 建树：初始化区间 [l, r]，所有格子颜色为1
void build(int idx, int l, int r) {
    tree[idx].cols = 1;         // 仅颜色1，掩码为 1 << 0 = 1
    tree[idx].lazy = 0;         // 无懒标记
    if (l == r) return;
    int mid = (l + r) >> 1;
    build(idx << 1, l, mid);
    build(idx << 1 | 1, mid + 1, r);
}

// 将懒标记下传给左右儿子
inline void pushDown(int idx) {
    if (tree[idx].lazy != 0) {
        int c = tree[idx].lazy;
        int left = idx << 1, right = idx << 1 | 1;
        tree[left].lazy = c;
        tree[left].cols = 1 << (c - 1);   // 只有一个颜色
        tree[right].lazy = c;
        tree[right].cols = 1 << (c - 1);
        tree[idx].lazy = 0;               // 清除当前节点懒标记
    }
}

// 区间涂色：将 [L, R] 染成颜色 c
```

```
void update(int idx, int l, int r, int L, int R, int c) {
    if (L <= l && r <= R) {
        // 当前区间完全被覆盖
        tree[idx].lazy = c;
        tree[idx].cols = 1 << (c - 1);
        return;
    }
    pushDown(idx);
    int mid = (l + r) >> 1;
    if (L <= mid) update(idx << 1, l, mid, L, R, c);
    if (R > mid) update(idx << 1 | 1, mid + 1, r, L, R, c);
    // 合并左右儿子的颜色集合
    tree[idx].cols = tree[idx << 1].cols | tree[idx << 1 | 1].cols;
}

// 区间查询：返回 [L, R] 的颜色掩码
int query(int idx, int l, int r, int L, int R) {
    if (L <= l && r <= R) {
        return tree[idx].cols;
    }
    pushDown(idx);
    int mid = (l + r) >> 1;
    int res = 0;
    if (L <= mid) res |= query(idx << 1, l, mid, L, R);
    if (R > mid) res |= query(idx << 1 | 1, mid + 1, r, L, R);
    return res;
}

int main() {
    freopen("color.in", "r", stdin);
    freopen("color.out", "w", stdout);

    cin >> L >> T >> O;
    build(1, 1, L);

    while (O--) {
        char op;
```

```
        cin >> op;
        if (op == 'C') {
            int A, B, C;
            cin >> A >> B >> C;
            if (A > B) swap(A, B);    // 保证 A <= B
            update(1, 1, L, A, B, C);
        } else { // op == 'P'
            int A, B;
            cin >> A >> B;
            if (A > B) swap(A, B);
            int ansMask = query(1, 1, L, A, B);
            // 统计二进制中1的个数
            cout << __builtin_popcount(ansMask) << "\n";
        }
    }

    return 0;
}
```

总结:

- **暴力** 实现简单，但 $O(O \cdot L)$ 无法应对 10^5 数据规模。
- **分块** 在块大小 $\approx \sqrt{L}$ 时达 $O(O\sqrt{L})$，平衡了实现难度与效率，足以通过本题。
- **线段树 + 状态压缩** 充分利用 $T \leq 30$ 的条件，用位运算加速合并，复杂度最优 $O(O \log L)$，适合作为标准解法。