**题目大意**

给定长度为 $n$ 的数组 $A$ 以及阈值 $m$。要求支持区间减去 $X$、区间元素除以 2 向下取整两种操作。若某次操作后数组最小值小于等于 $m$，则全局重置次数 $k$ 增加 1，并将数组完全恢复为初始状态。要求输出最终的 $k$ 值。

**超时原因分析**

在此前的思路中，使用了势能线段树（区间最值配合区间减法代替除法）。但本题具有在常数时间内触发时空回溯的机制。一旦触发回溯，线段树瞬间恢复初始极差（即势能回满）。如果遇到恶意构造的数据，频繁触发重置并交替执行区间除法，会导致每次除法都因为区间内极差较大而递归至叶子节点，最终单次操作时间退化，整体时间复杂度达到 $O(Q \times n)$，从而导致超时。

**复合函数与懒惰标记优化**

为了解决势能退化问题，我们需要寻找能够支持严格对数级别复杂度区间更新的算法。观察本题的两个操作：减法操作可以表示为 $f(x) = x - X$；除法操作可以表示为 $g(x) = \lfloor \frac{x}{2} \rfloor$。这两个函数都严格单调不减，因此区间的最小值经过若干次操作后，必然等于初始最小值的对应操作结果。我们在线段树中只需维护区间最小值，无需关注最大值或极差。

任何减法和除法操作的组合，都可以被化简为一个复合函数：

$$h(x) = \left\lfloor \frac{x - A}{2^k} \right\rfloor$$

其中 $A$ 为累计的减法偏移量，$k$ 为累计的右移次数。

当我们给带有懒惰标记 $(A_{old}, k_{old})$ 的节点施加新的操作标记 $(A_{new}, k_{new})$ 时，复合后的函数为：

$$h_{new}(h_{old}(x)) = \left\lfloor \frac{\left\lfloor \frac{x - A_{old}}{2^{k_{old}}} \right\rfloor - A_{new}}{2^{k_{new}}} \right\rfloor$$

由于 $A_{new}$ 是一个整数，我们可以利用向下取整的性质 $\lfloor y \rfloor - C = \lfloor y - C \rfloor$（其中 $C$ 为整数），将 $A_{new}$ 移入内层的向下取整符号中并通分：

$$= \left\lfloor \frac{\left\lfloor \frac{x - A_{old} - A_{new} \times 2^{k_{old}}}{2^{k_{old}}} \right\rfloor}{2^{k_{new}}} \right\rfloor$$

紧接着，利用向下取整除法的另一个核心嵌套性质 $\left\lfloor \frac{\lfloor y/a \rfloor}{b} \right\rfloor = \lfloor \frac{y}{ab} \rfloor$（其中 $a, b$ 为正整数），我们将两层嵌套的向下取整合并为一层：

$$= \left\lfloor \frac{x - A_{old} - A_{new} \times 2^{k_{old}}}{2^{k_{old} + k_{new}}} \right\rfloor$$

由此我们可以得到常数时间复杂度下的懒惰标记合并公式：

$$A' \leftarrow A_{old} + A_{new} \times 2^{k_{old}}$$

$$k' \leftarrow k_{old} + k_{new}$$

**补充：向下取整除法的奇妙性质**

一个很自然的疑问是：公式 $h(x) = \lfloor \frac{x - A}{2^k} \rfloor$ 形式上看起来像是“先减去 $A$，再除以 $2^k$”，那么如果是“先除以 $2^k$，再减去一个数”，还能用这个公式表示吗？

答案是**完全可以**。这得益于向下取整函数的一个重要性质：对于任意实数 $y$ 和任意**整数** $C$，都有：

$$\lfloor y \rfloor - C = \lfloor y - C \rfloor$$

所以，如果我们先对 $x$ 执行除法 $\lfloor \frac{x}{2^k} \rfloor$，然后再减去一个整数 $X$，其结果为：

$$\left\lfloor \frac{x}{2^k} \right\rfloor - X = \left\lfloor \frac{x}{2^k} - X \right\rfloor = \left\lfloor \frac{x - X \times 2^k}{2^k} \right\rfloor$$

这正好又回到了 $\left\lfloor \frac{x - A'}{2^{k'}} \right\rfloor$ 的标准形式（其中 $A' = X \times 2^k$）。因此，无论“先减后除”还是“先除后减”，由于减去的总是整数，都可以被完美地统一在这个单一的复合函数模型中！这也是本题懒惰标记能够 $O(1)$ 合并的数学基石。

**直观理解：嵌套向下取整的合并**

我们在推导中还用到了 $\left\lfloor \frac{\lfloor y/a \rfloor}{b} \right\rfloor = \lfloor \frac{y}{ab} \rfloor$ 这个性质。抛开繁琐的公式证明，我们可以这样极其直观地来理解它：

假设你有 $y$ 颗糖果，要平均分给 $a \times b$ 个小朋友。左边的分法 $\left\lfloor \frac{\lfloor y/a \rfloor}{b} \right\rfloor$ 是：先将所有糖果按每 $a$ 颗装成一个“小包”，剩下不够 $a$ 颗的散装糖果直接扔掉不计（对应 $\lfloor y/a \rfloor$）。然后再把这些“小包”每 $b$ 包进一步装进一个“大箱子”，不够 $b$ 包的也直接扔掉。右边的分法 $\lfloor \frac{y}{ab} \rfloor$ 是：直接把所有糖果按每 $a \times b$ 颗装进一个“大箱子”，剩下的全扔掉。

很显然，不管你是分两步“先打包再装箱”，还是直接一步到位“按总数装箱”，最终得到的**完整大箱子的数量绝对是一模一样的**！因为一个大箱子无论如何都需要实打实的 $a \times b$ 颗糖果。在第一步里扔掉的散装糖果（最多 $a-1$ 颗），加上第二步里扔掉的零散小包（最多 $b-1$ 包，折合 $(b-1) \times a$ 颗糖果），把这些丢弃的边角料全部加起来，总数最多也只有 $(a-1)+(b-1) \times a = a \times b - 1$ 颗糖果，绝对凑不够装满一个新的大箱子。既然所有丢弃的边角料加起来都凑不出一箱，那自然对大箱子的总数不会产生任何实质性影响了。这就是为什么两层向下取整可以直接无脑拍扁合并为一层的原因！

**复杂度与边界处理**

时间复杂度：去除了势能线段树的极差分类讨论，每次区间修改和查询严格受限于线段树的深度。因此单次操作的时间复杂度为严格的 $O(\log n)$，整体时间复杂度为 $O(n + Q \log n)$。

空间复杂度：仅需记录常量个状态，为 $O(n)$。

数值溢出处理：因为 $m \geq 0$，任何元素在被除以 2 约 30 次后必定小于等于阈值触发重置，因此 $k \leq 30$。然而，在标记合并时 $A_{new} \times 2^{k_{old}}$ 的累加可能会使 $A$ 超过六十四位整数的表示范围。我们需要使用一百二十八位整数（即代码中的 ）来存储 $A$，以防止在累加偏移量时发生溢出。并且在处理负数位移时，手写了向下取整的逻辑，以防语言标准的特性导致计算出现偏差。

**核心代码**

```cpp
#include <iostream>

using namespace std;

const int MAXN = 100005;

struct Node {
    int l, r;
    long long min_v;
    __int128 lazy_A;
    int lazy_k;
    bool lazy_reset;
    long long init_min;
} tr[MAXN << 2];

long long A_arr[MAXN];
int n, q;
long long m;

void pushup(int u) {
    tr[u].min_v = min(tr[u << 1].min_v, tr[u << 1 | 1].min_v);
}

void apply_reset(int u) {
    tr[u].min_v = tr[u].init_min;
    tr[u].lazy_A = 0;
    tr[u].lazy_k = 0;
    tr[u].lazy_reset = true;
}

void apply_tag(int u, __int128 A_new, int k_new) {
    __int128 val = tr[u].min_v;
    val -= A_new;
    // 兼容 C++ 中负数向零取整的区别，使用严谨的向下取整逻辑
    if (val < 0 && k_new > 0) {
```

```
        __int128 divisor = ((__int128)1 << k_new);
        __int128 res = val / divisor;
        if (val % divisor != 0) {
            res -= 1;
        }
        val = res;
    } else {
        val >>= k_new;
    }
    tr[u].min_v = (long long)val;

    tr[u].lazy_A += A_new * ((__int128)1 << tr[u].lazy_k);
    tr[u].lazy_k += k_new;
}

void pushdown(int u) {
    if (tr[u].lazy_reset) {
        apply_reset(u << 1);
        apply_reset(u << 1 | 1);
        tr[u].lazy_reset = false;
    }
    if (tr[u].lazy_A != 0 || tr[u].lazy_k != 0) {
        apply_tag(u << 1, tr[u].lazy_A, tr[u].lazy_k);
        apply_tag(u << 1 | 1, tr[u].lazy_A, tr[u].lazy_k);
        tr[u].lazy_A = 0;
        tr[u].lazy_k = 0;
    }
}

void build(int u, int l, int r) {
    tr[u].l = l; tr[u].r = r;
    tr[u].lazy_A = 0;
    tr[u].lazy_k = 0;
    tr[u].lazy_reset = false;
    if (l == r) {
        tr[u].min_v = tr[u].init_min = A_arr[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(u << 1, l, mid);
    build(u << 1 | 1, mid + 1, r);
    pushup(u);
    tr[u].init_min = tr[u].min_v;
}

void modify_sub(int u, int l, int r, long long x) {
    if (tr[u].l >= l && tr[u].r <= r) {
        apply_tag(u, x, 0);
        return;
    }
    pushdown(u);
    int mid = (tr[u].l + tr[u].r) >> 1;
    if (l <= mid) modify_sub(u << 1, l, r, x);
    if (r > mid) modify_sub(u << 1 | 1, l, r, x);
    pushup(u);
}

void modify_div(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) {
        apply_tag(u, 0, 1);
        return;
    }
    pushdown(u);
```

```
    int mid = (tr[u].l + tr[u].r) >> 1;
    if (l <= mid) modify_div(u << 1, l, r);
    if (r > mid) modify_div(u << 1 | 1, l, r);
    pushup(u);
}

void solve() {
    cin >> n >> m >> q;
    for (int i = 1; i <= n; i++) {
        cin >> A_arr[i];
    }
    build(1, 1, n);

    int k_ans = 0;
    for (int i = 0; i < q; i++) {
        int op, l, r;
        cin >> op >> l >> r;
        if (op == 1) {
            long long x;
            cin >> x;
            modify_sub(1, l, r, x);
        } else {
            modify_div(1, l, r);
        }

        if (tr[1].min_v <= m) {
            k_ans++;
            apply_reset(1);
        }
    }
    cout << k_ans << "\n";
}

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

    int T;
    if (cin >> T) {
        while (T--) {
            solve();
        }
    }
    return 0;
}
```