1007 喂马——题解（AI 辅助版）

znzryb

April 19, 2026

1007 喂马

Time Limit: 20000 / 10000 MS (Java / Others)
Memory Limit: 524288 / 524288 K (Java / Others)

Problem Description

2026 丙午马年，某马术俱乐部有 n 匹马排成一排，编号 $1, 2, \ldots, n$。俱乐部设有 m 个饲料槽，编号 $1, 2, \ldots, m$。初始时，第 i 匹马被分配到第 a_i 号饲料槽（$a_i = 0$ 表示该马暂未分配）。每个饲料槽有一个投喂量计数 b_i，初始均为 0。

你需要依次执行 q 次操作，操作有两种类型：

① 给定 l, r, x，将编号在 $[l, r]$ 内的所有马重新分配到第 x 号饲料槽。

② 给定 l, r, x，对编号在 $[l, r]$ 内的每匹马，若该马已分配饲料槽（即 $a_i \neq 0$），则向其饲料槽投喂 x 单位饲料（即 b_{a_i} 加上 x）。

请输出所有操作执行完毕后，每个饲料槽的总投喂量。

数据范围

$1 \le T \le 5$，$1 \le n, m, q \le 2 \times 10^6$，$0 \le a_i \le m$，$opt \in \{1, 2\}$，$1 \le l \le r \le n$，$1 \le x \le m$。所有测试数据的 n 与 q 之和均不超过 2×10^6。

1 核心思路：把投喂量先记在马身上

注意到操作 ② 里，真正影响答案的，是” 这匹马当前属于哪个槽”。如果一匹马在处理操作 ② 时属于槽 c，那这次投喂的 x 就加到 `b[c]` 上。

一个关键的观察：**操作 ② 本身不需要知道颜色**，它只是在 $[l, r]$ 上加一个值 x——不管这段里的马属于哪个槽。我们可以先把这些” 投喂” 全部记在马的编号上，开一个 BIT（支持区间加、区间查询），操作 ② 直接对 $[l, r]$ 区间加 x，BIT 里 `S[i]` 的含义就是” 马 i 被操作过的所有投喂量之和”。

```
// 操作\optwo{} 直接变成一次 BIT 区间加，O(log n)，完全不碰 ODT
if (opt == 2) bit.add(l, r, x);
```

剩下的问题是：这些积在马身上的投喂量，怎么最终归到正确的槽？

2 在换槽的瞬间结账

马的槽会被操作 ① 改变。想清楚下面这件事：一匹马 i 一开始属于槽 c_{old}，然后某次操作 ① 把它改成槽 c_{new}。在这次改变发生之前，所有打到马 i 上的投喂量，都应该归给 c_{old}。

所以在执行操作 ① 的那一刻，对于每一段被覆盖的旧区间 $[L, R]$（旧颜色 c），把 BIT 里 $[L, R]$ 的区间和查出来，直接加到 `b[c]` 上就行了：

```
// 在 assign 里遍历每一个将被推平的旧段 [L, R, c]
ull V = bit.query(L, R);   // 这段马身上目前积累的总投喂量
if (c != 0) b[c] += V;     // 结账给旧槽
```

但光加还不够——这段马接下来要属于新槽 c_{new}。假设之后又有操作 ② 打到它们身上，那些投喂量不应该再算到旧槽里去了。所以还要把”当前已积累的基准量”从新槽里提前减掉：

```
if (x != 0) b[x] -= V;    // 让新槽从"零基准"开始计，而不是从当前 V 开始
```

这样做之后，将来再查 $[L, R]$ 的区间和时，那个值只包含”进入新槽 x 之后新增的投喂量”，正好是应该归给槽 x 的部分。这就是整个方案的核心：**先减后加，差分结账**。

3 最后一次结账

所有操作结束后，ODT 里剩下的每个段 $[L, R, c]$，它们在最后一次换颜色之后可能又被投喂了若干次，这些投喂量还留在 BIT 里没有结算。遍历一遍 ODT，对每个非零颜色的段，把区间和加到对应的 `b[c]` 上：

```
for (auto &nd : odt) {
    if (nd.c != 0) b[nd.c] += bit.query(nd.l, nd.r);
}
```

4 为什么这样做 ODT 不会炸

操作 ② 根本不碰 ODT，只动 BIT，每次 $O(\log n)$。ODT 只在操作 ① 里被动，而每次操作 ① 在推平时会删掉旧段、插入新段。ODT 的快就是因为”被删的段不会再回来”——整个生命周期里最多插入 $n+q$ 个段，所有操作 ① 加在一起最多删 $n+q$ 次，每次删的时候顺便做一次 BIT 区间查询。总代价完全可控。

5 处理 `a[i] = 0` 的马

颜色为 0 代表未分配。在 assign 遍历时：

```
if (c != 0) b[c] += V;    // c == 0 时直接跳过，不给任何槽结账
if (x != 0) b[x] -= V;    // x == 0 时也跳过
```

颜色为 0 的段在换出去时不结账、换进来时也不做减操作。于是这段马在”属于槽 0”期间积累的投喂量就被彻底丢弃，符合题意。

6 溢出问题

x 最大 2×10^6，区间最大 2×10^6，操作最多 2×10^6 次，最大总量接近 `long long` 的上限。而差分结账时 `b[x] -= V` 会让 `b[x]` 暂时变成负数（在 `long long` 里意味着有溢出风险）。

改用 `unsigned long long` 就没这个问题——所有加减在模 2^{64} 的环里转，最终结果只要是正确的正数就能正常打印。

7 完整代码

```
#include <iostream>
#include <vector>
#include <set>
using namespace std;

typedef unsigned long long ull;
const int MAXN = 2000005;

// BIT: 支持区间加、区间查询
// 维护两个差分数组 d1[i] 和 d2[i] = i*d1[i]
// 前缀和 = (p+1)*sum(d1, p) - sum(d2, p)
ull bit1[MAXN], bit2[MAXN];
ull b[MAXN];
int a[MAXN];
```

```cpp
int n, m, q;

void bit_add(int idx, ull v1, ull v2) {
  for (; idx <= n; idx += idx & -idx) {
    bit1[idx] += v1;
    bit2[idx] += v2;
  }
}
void range_add(int l, int r, ull val) {
  bit_add(l,     val,          (ull)l * val);
  bit_add(r + 1, 0ULL - val,   0ULL - (ull)(r + 1) * val);
}
ull prefix_sum(int idx) {
  ull s1 = 0, s2 = 0, k = idx;
  for (; idx > 0; idx -= idx & -idx) { s1 += bit1[idx]; s2 += bit2[idx]; }
  return (k + 1) * s1 - s2;
}
ull range_query(int l, int r) {
  return prefix_sum(r) - prefix_sum(l - 1);
}

// ODT 节点: [l, r] 内所有马属于槽 c
struct Node {
  int l, r;
  mutable int c;
  Node(int l_, int r_ = 0, int c_ = 0) : l(l_), r(r_), c(c_) {}
  bool operator<(const Node& o) const { return l < o.l; }
};
set<Node> odt;

auto split(int x) {
  if (x > n) return odt.end();
  auto it = --odt.upper_bound(Node(x));
  if (it->l == x) return it;
  int l = it->l, r = it->r, c = it->c;
  odt.erase(it);
  odt.insert(Node(l, x - 1, c));
  return odt.insert(Node(x, r, c)).first;
}

void assign_color(int l, int r, int x) {
  auto itr = split(r + 1);
  auto itl = split(l);
  for (auto it = itl; it != itr; ++it) {
    int L = it->l, R = it->r, c = it->c;
    ull V = range_query(L, R);
    if (c != 0) b[c] += V;   // 旧槽结账
    if (x != 0) b[x] -= V;   // 新槽扣除已积累的基准量
  }
  odt.erase(itl, itr);
  odt.insert(Node(l, r, x));
}

// 快读快写
namespace FastIO {
  const int BUFSZ = 1 << 20;
  char ibuf[BUFSZ], obuf[BUFSZ];
  char *ip1 = ibuf, *ip2 = ibuf;
  int opos = 0;
  inline char gc() {
    if (ip1 == ip2) {
      ip2 = (ip1 = ibuf) + fread(ibuf, 1, BUFSZ, stdin);
      if (ip1 == ip2) return EOF;
```

```
    }
    return *ip1++;
  }
  inline void pc(char c) {
    if (opos == BUFSZ) { fwrite(obuf, 1, BUFSZ, stdout); opos = 0; }
    obuf[opos++] = c;
  }
  template<typename T> inline void rd(T &x) {
    x = 0; char c = gc();
    while (c < '0' || c > '9') { if (c == EOF) return; c = gc(); }
    while (c >= '0' && c <= '9') { x = x * 10 + c - '0'; c = gc(); }
  }
  template<typename T> inline void wr(T x) {
    if (x == 0) { pc('0'); return; }
    static char st[24]; int top = 0;
    while (x) { st[top++] = x % 10 + '0'; x /= 10; }
    while (top--) pc(st[top]);
  }
  inline void flush() { if (opos) { fwrite(obuf, 1, opos, stdout); opos = 0; } }
}
using namespace FastIO;

void solve() {
  rd(n); rd(m); rd(q);
  for (int i = 0; i <= n + 1; ++i) bit1[i] = bit2[i] = 0;
  for (int i = 0; i <= m + 1; ++i) b[i] = 0;
  odt.clear();

  for (int i = 1; i <= n; ++i) rd(a[i]);

  // 初始 ODT：相邻同色的马合并成一段
  for (int i = 1; i <= n; ) {
    int j = i;
    while (j <= n && a[j] == a[i]) ++j;
    odt.insert(Node(i, j - 1, a[i]));
    i = j;
  }

  for (int i = 0; i < q; ++i) {
    int opt, l, r; ull x;
    rd(opt); rd(l); rd(r); rd(x);
    if (opt == 1) assign_color(l, r, (int)x);
    else          range_add(l, r, x);   // 操作\optwo{}：只动 BIT，不碰 ODT
  }

  // 最终结算：ODT 里每个残留段的积累量归给对应槽
  for (auto &nd : odt)
    if (nd.c != 0) b[nd.c] += range_query(nd.l, nd.r);

  for (int i = 1; i <= m; ++i) {
    wr(b[i]);
    pc(i == m ? '\n' : ' ');
  }
}

int main() {
  int T; rd(T);
  while (T--) solve();
  flush();
  return 0;
}
```

8 把样例跟着走一遍

初始：`a = [1, 0, 3, 2, 4, 1]`，ODT 各段：`[1,1,1] [2,2,0] [3,3,3] [4,4,2] [5,5,4] [6,6,1]`，`S = [0,0,0,0,0,0]`，`b = [0,0,0,0]`。

1. **操作 2 1 4 3**：BIT 在 $[1,4]$ 加 3，`S = [3,3,3,3,0,0]`。
2. **操作 1 2 5 1**：把 $[2,5]$ 推成颜色 1。遍历旧段结账：
   - `[2,2,0]`，$V=3$，`c=0` 跳过；`b[1] -= 3`。
   - `[3,3,3]`，$V=3$，`b[3] += 3`；`b[1] -= 3`。
   - `[4,4,2]`，$V=3$，`b[2] += 3`；`b[1] -= 3`。
   - `[5,5,4]`，$V=0$，无影响。

   此时 `b = [-9, 3, 3, 0]`（`ull` 环里），ODT 变为 `[1,1,1] [2,5,1] [6,6,1]`。
3. **操作 2 1 6 2**：BIT 在 $[1,6]$ 加 2，`S = [5,5,5,5,2,2]`。
4. **操作 1 3 4 4**：把 $[3,4]$ 推成颜色 4。split 出 `[3,4,1]`，$V=10$，`b[1] += 10`，`b[4] -= 10`。`b = [1, 3, 3, -10]`，ODT 变为 `[1,1,1] [2,2,1] [3,4,4] [5,5,1] [6,6,1]`。
5. **操作 2 2 5 4**：BIT 在 $[2,5]$ 加 4，`S = [5,9,9,9,6,2]`。
6. **最终结算**：
   - `[1,1,1]`：`b[1] += 5`。
   - `[2,2,1]`：`b[1] += 9`。
   - `[3,4,4]`：`b[4] += 18`。
   - `[5,5,1]`：`b[1] += 6`。
   - `[6,6,1]`：`b[1] += 2`。

   `b[1] = 1+5+9+6+2 = 23`，`b[2] = 3`，`b[3] = 3`，`b[4] = -10+18 = 8`。

输出 `23 3 3 8`，与样例答案一致。