线段树 （区间加 + 区间和）

# 一、问题回顾

给定一个长度为 `n` 的数列 `a`，支持以下两种操作：

1. **区间加法**：将区间 `[l, r]` 内每个元素加上 `k`
2. **区间求和**：查询区间 `[l, r]` 内所有元素的和

约束条件：

- `1 ≤ n, m ≤ 1e5`
- 任意时刻区间和 ≤ `2 × 10^18`

👉 典型的 **线段树 + 懒惰标记（Lazy Propagation）** 模板题。

---

# 二、核心数据结构设计

```
vector<long long> tree; // 线段树，维护区间和
vector<long long> lazy; // 懒惰标记，表示“整段待加的值”
int n;                  // 数组长度
```

- `tree[node]`：表示当前节点区间的 **真实区间和**
- `lazy[node]`：表示该区间的子节点尚未下推的增量

---

# 三、建树（Build）

```
void build(int node, int start, int end, vector<int>& arr)
```

## 思路

- 如果是叶子节点：直接赋值
- 否则递归构建左右子树，再合并

## 关键点

- 线段树使用 **1 作为根节点编号**
- 区间采用 **0-based**（`[0, n-1]`）

---

# 四、懒惰标记下推（pushDown）

```
void pushDown(int node, int start, int end)
```

### 作用

当当前区间存在 `lazy[node]` 时：

- 把增量分发给左右子区间
- 更新子节点的区间和
- 清空当前节点的懒标记

### 关键实现细节（非常重要）

```
tree[leftChild]  += lazy[node] * (long long)(mid - start + 1);
tree[rightChild] += lazy[node] * (long long)(end - mid);
```

📌 **必须显式转为 `long long`**，否则在大数据下会整数溢出，导致部分点 WA。

---

## 五、区间更新（区间加）

```
void updateRange(int node, int start, int end, int L, int R, long long val)
```

### 三种情况

1. **完全不相交**：直接返回
2. **完全包含**：
   - 更新当前区间和
   - 累加懒惰标记
3. **部分重叠**：
   - 先下推懒标记
   - 递归更新左右子区间
   - 回溯时重新计算区间和

### 核心语句

```
tree[node] += val * (long long)(end - start + 1);
lazy[node] += val;
```

---

## 六、区间查询（区间和）

```
long long query(int node, int start, int end, int L, int R)
```

### 查询逻辑

- 不相交：返回 0
- 完全覆盖：直接返回 `tree[node]`
- 部分覆盖：
  - 先 `pushDown`
  - 查询左右区间并合并

---

## 七、下标处理（易错点）

题目输入是 **1-based 区间**，而线段树内部使用 **0-based**。

```
st.updateRange(x - 1, y - 1, k);
st.query(x - 1, y - 1);
```

📌 **下标转换必须在 `main` 中完成，线段树内部保持统一规范。**

---

## 八、IO 性能优化（防止 TLE）

```
ios::sync_with_stdio(false);
cin.tie(nullptr);
```

- 禁止 `endl`（会强制刷新缓冲区）
- 使用 `\n` 输出

这是 P3372 出现"只剩 1~2 个点不过"的常见原因。

---

## 九、时间与空间复杂度

- 单次操作：`O(log n)`
- 总时间复杂度：`O(m log n)`
- 空间复杂度：`O(n)`（线段树约 `4n`）

---

## 十、总结

- 本题是 **线段树懒惰标记的标准模板**
- 核心难点不在算法，而在：
  - 下标统一
  - `long long` 溢出处理

- IO 性能优化

> 掌握本题后，可直接进阶：
>
> - P3373（区间乘 + 加 + 取模）
> - 动态开点线段树
> - 扫描线 + 线段树

---

示例代码

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

class SegmentTreeLazy
{
private:
    vector<long long> tree; // 线段树数组（区间和）
    vector<long long> lazy; // 懒惰标记数组
    int n;

    // 构建线段树
    void build(int node, int start, int end, vector<int>& arr)
    {
        if (start == end)
        {
            tree[node] = arr[start];
        }
        else
        {
            int mid = (start + end) / 2;
            int leftChild = 2 * node;
            int rightChild = 2 * node + 1;

            build(leftChild, start, mid, arr);
            build(rightChild, mid + 1, end, arr);

            tree[node] = tree[leftChild] + tree[rightChild];
        }
    }

    // 下推懒标记
    void pushDown(int node, int start, int end)
    {
        if (lazy[node] != 0)
        {
            int mid = (start + end) / 2;
            int leftChild = 2 * node;
            int rightChild = 2 * node + 1;

            // 关键修复：强制转换为 long long 再计算
            tree[leftChild] += lazy[node] * (long long)(mid - start + 1);
```

```
            lazy[leftChild] += lazy[node];

            tree[rightChild] += lazy[node] * (long long)(end - mid);
            lazy[rightChild] += lazy[node];

            // 清除当前节点的懒标记
            lazy[node] = 0;
        }
    }

    // 区间更新：给区间 [L, R] 的每个元素加上 val
    void updateRange(int node, int start, int end, int L, int R, long long val)
    {
        if (R < start || L > end)
        {
            // 区间完全不重叠
            return;
        }

        if (L <= start && end <= R)
        {
            // 区间完全包含，打上懒标记
            // 关键修复：强制转换为 long long 再计算
            tree[node] += val * (long long)(end - start + 1);
            lazy[node] += val;
            return;
        }

        // 区间部分重叠，需要下推懒标记
        pushDown(node, start, end);

        int mid = (start + end) / 2;
        int leftChild = 2 * node;
        int rightChild = 2 * node + 1;

        updateRange(leftChild, start, mid, L, R, val);
        updateRange(rightChild, mid + 1, end, L, R, val);

        // 更新当前节点
        tree[node] = tree[leftChild] + tree[rightChild];
    }

    // 区间查询
    long long query(int node, int start, int end, int L, int R)
    {
        if (R < start || L > end)
        {
            return 0;
        }

        if (L <= start && end <= R)
        {
            return tree[node];
        }
```

```
        // 需要访问子节点，先下推懒标记
        pushDown(node, start, end);

        int mid = (start + end) / 2;
        int leftChild = 2 * node;
        int rightChild = 2 * node + 1;

        long long sumLeft = query(leftChild, start, mid, L, R);
        long long sumRight = query(rightChild, mid + 1, end, L, R);

        return sumLeft + sumRight;
    }

public:
    SegmentTreeLazy(vector<int> &input)
    {
        n = input.size();
        tree.resize(4 * n, 0);
        lazy.resize(4 * n, 0);
        build(1, 0, n - 1, input);
    }

    // 公开接口：区间更新（区间加）
    void updateRange(int L, int R, long long val)
    {
        updateRange(1, 0, n - 1, L, R, val);
    }

    // 公开接口：区间查询
    long long query(int L, int R)
    {
        return query(1, 0, n - 1, L, R);
    }
};

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

    int n, m;
    cin >> n >> m;

    vector<int> a(n);

    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    SegmentTreeLazy st(a);

    for (int i = 0; i < m; i++)
```

```
    {
        int op;
        cin >> op;

        if (op == 1)
        {
            int x, y;
            long long k;
            cin >> x >> y >> k;
            st.updateRange(x - 1, y - 1, k);
        }
        else
        {
            int x, y;
            cin >> x >> y;
            cout << st.query(x - 1, y - 1) << '\n';
        }
    }
    return 0;
}
```