树状数组和线段树

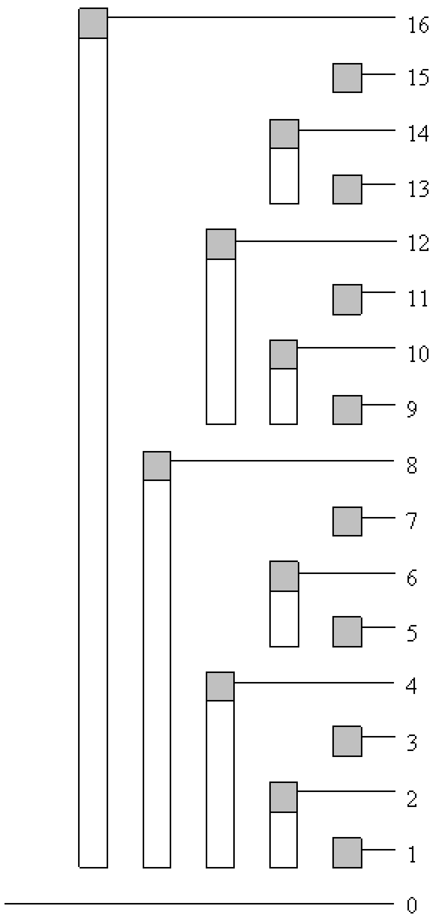
树状数组和线段树都可以用来维护数列信息，并且支持单点/区间修改，单点/区间查询。

一、树状数组

问题思考：给出一个数组，我们需要执行以下两种操作：

- 单点更新：修改数组中的某一个数字。
- 区间求和：求某一个区间内的数字之和。

对于问题1，我们通常可以在$O(1)$的时间内解决，对于问题二，我们通常可以在$O(n)$的时间内解决。如果我们使用前缀和解决问题二，我们可以在$O(1)$的时间内求解前缀和，但是我们修改数组元素时，则要重新计算一次前缀和数组$O(n)$。假设我们有m次查询，对于朴素算法，总体的时间复杂度就达到了$O(nm)$,当数据量较大时，这是不可接受的，对此类问题我们可以使用树状数组或线段树将时间复杂度降低到$O(mlogn)$。



如图可见，对于树状数组，编号为x的节点，存储着区间$[x - lowbit(x), x]$的信息，x的父亲节点的编号则是$x + lowbit(x)$，每个节点覆盖的长度就是$lowbit(x)$。
注：$lowbit(x) = x\& - x$; // 代表整数x在二进制表示下的最后一位1代表的数。

利用差分思想，将一个数表示为多数之和的形式，在查询时，将区间分成若干小块儿进行求和。现在假设我们想求前13个元素的和，13的二进制表示为1101，树状数组的每个节点x存储着区间$[x - lowbit(x), x]$的信息

$$sum(0,13) = sum(0,1101_2) = tr[1101_2] + tr[1100_2] + tr[1000_2]$$

恰好包含了13的二进制表示中依次去除每一个1。

这里对$lowbit$函数进行解释

假设 x的二进制表示为：$x = A1B_2$,

A表示一个01组合，B表示一个任意长度的全是0的组合。$A^{-1}0B^{-1}$

而$-x$的二进制表示为: $-x = A^{-1}0B^{-1}$

因此$x\&-x = (000)_a1(000)_b$，其中，$a = length(A), b = length(B)$

树状数组建树（单点修改）：

基于差分和前缀和思想，对于区间和问题，在初始化序列（或对序列进行单点修改）时，我们将编号为x的点加上c，在一次输入结束后完成建树（修改）。

```
void add(int x, int c)
{
    for (int i = x; i <= n; i += lowbit(i))
    {
        tr[i] += c;
    }
}
```

前缀和求解：（区间查询）
假设我们想求前x个元素的前缀和，我们执行以下操作：

- 将$tr[x]$的值加入到答案中
- 将x的值减去$tree[x]$包含的元素个数$lowbit(x)$
- 重复上述操作直至$x < 0$

```
int sum(int x)
{
    int rs = 0;
    for (int i = x; i; i -= lowbit(i))
    {
        rs += tr[i];
    }
    return rs;
}
```

单点查询：

通常有两种方法：

- 方法一是，在更新节点信息时，利用额外空间维护一个A序列信息，这样我们可以在$O(1)$的时间内求解。
- 方法二是，$tr[x] = sum(x) - sum(x-1)$; // 前$x$个元素的和减去前$x-1$个元素的和。//当然我们也可以使用这种方法求区间$(i, j)$的和。

区间修改(给某一区间加上相同的值)

```
void update(int x, int c)
{
    for (int i = x; i <= n; i += lowbit(i))
    {
        tr[i] += c;
    }
}
void add(int l, int r, int x)
{
    update(l, x);
    update(r+1, -x);
}
```

一个简单的整数问题

给定长度为 N 的数列 A，然后输入 M 行操作指令。

第一类指令形如 `C l r d`，表示把数列中第 l~r 个数都加 d。

第二类指令形如 `Q x`，表示询问数列中第 x个数的值。

对于每个询问，输出一个整数表示答案。

`输入格式`

第一行包含两个整数N 和 M。

第二行包含 N 个整数 A[i]。

接下来 M 行表示 M 条指令，每条指令的格式如题目描述所示。

`输出格式`

对于每个询问，输出一个整数表示答案。

每个答案占一行。

`数据范围`

$$1 \leq N, M \leq 105, |d| \leq 10000, |A[i]| \leq 109$$

`输入样例：`

```
10 5
1 2 3 4 5 6 7 8 9 10
Q 4
Q 1
Q 2
C 1 6 3
Q 2
```

`输出样例：`

```
4
1
2
5
```

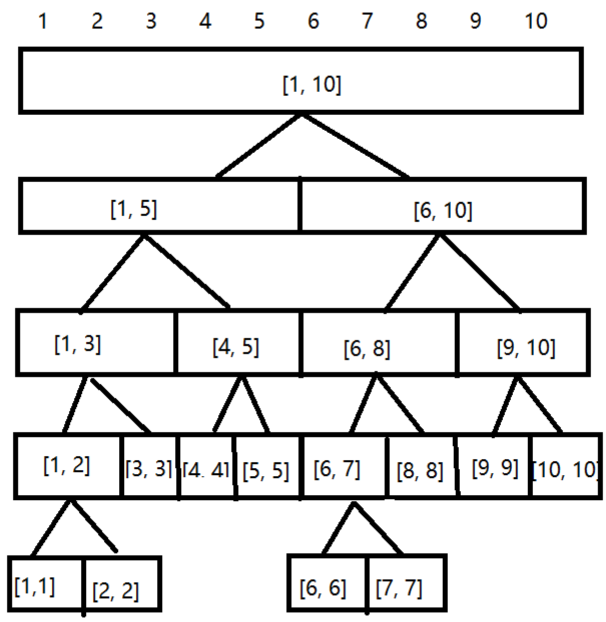
```
#include <cstdio>
```

```cpp
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

typedef long long LL;

const int N = 100010;

int n, m;
int a[N];
LL tr[N];

int lowbit(int x)
{
    return x & -x;
}

void add(int x, int c)
{
    for (int i = x; i <= n; i += lowbit(i)) tr[i] += c;
}

LL sum(int x)
{
    LL res = 0;
    for (int i = x; i; i -= lowbit(i)) res += tr[i];
    return res;
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++ ) scanf("%d", &a[i]);

    for (int i = 1; i <= n; i ++ ) add(i, a[i] - a[i - 1]);

    while (m -- )
    {
        char op[2];
        int l, r, d;
        scanf("%s%d", op, &l);
        if (*op == 'C')
        {
            scanf("%d%d", &r, &d);
            add(l, d), add(r + 1, -d);
        }
        else
        {
            printf("%lld\n", sum(l));
        }
    }

    return 0;
}
```

二、线段树

线段树是一种基于分治思想的二叉树结构，用于在区间上进行信息统计。与按照二进制位进行区间划分的树状数组相比，线段树是一种更加通用的数据结构，能维护的信息也更多，更灵活：

- 线段树的每个节点都代表一个区间。
- 线段树具有唯一的根节点。代表的区间是整个统计范围S。// [1, n]
- 线段树的每个叶子节点都代表一个长度为1的元区间[x, x]
- 对于每个内部节点[l, r]，它的左子节点是[l, mid], 它的右子节点是[mid + 1, r], 其中mid = (l + r) / 2; (向下取整)
- 需要满足：线段树维护的信息需要满足【区间加法】，以此将大问题划分成小问题。



（一）、简单线段树（不支持区间修改）

简单线段树支持：

- 单点修改
- 区间查询
- 单点查询

存储方式：

对于每个节点需要存储的信息有：

- 该节点所代表的区间边界
- 本区间所维护的信息 // 区间和，区间最值，区间`gcd`等等

```
struct Node
{
    int l, r; // 代表本届点维护的是区间 [l, r]的信息
    int info; // 线段树要维护的信息
}tr[N * 4]; // 要开四倍空间，课上解释为何开四倍空间
```

线段树建树：

```cpp
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r)
    {
        tr[u].info = a[l];
        return;
    }
    int mid = l + r >> 1; // >> 1 相当于整除以 2,
    build (u << 1, l, mid), build (u << 1 | 1, mid + 1, r);
    pushup(u); // 将线段树维护的信息向上更新
}
```

```cpp
// 对于pushup函数，这里我们以区间和问题为例

void pushup(int u)
{
    tr[u].info = tr[u << 1].info + tr[u << 1 | 1].info;
}
```

单点更新：

- 从根节点出发，寻找要修改位置所在的叶子节点
- 更新当前节点信息
- 向上更新信息

```cpp
void modify(int u, int x, int k) // 给a[x]加上k
{
    if (tr[u].l == x && tr[u].r == x)
    {
        tr[u].info += k;
        return;
    }
    int mid = tr[u].l + tr[u].r >> 1;
    if (x <= mid) modify(u << 1, x, k);
    else modify(u << 1 | 1, x, k);
    pushup(u);
}
```

区间查询：

```cpp
int query(int u, int l, int r) // 查询区间[l, r]的信息
{
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].info;
    if (tr[u].l > r || tr[u].r < l) return -inf; // 异常，数据正常时不会执行这一句代码
    int mid = tr[u].l + tr[u].r >> 1;
    int rs = 0;
    if (r <= mid ) rs = query(u << 1, l, r);
    if (l > mid) rs += query(u << 1 | 1, l, r);
    return rs;
}
```

我们练习一下：

A. Segment with the Maximum Sum

In this problem, you need to write a segment tree to find the segment with the maximum sum.

Input

The first line contains two numbers n and m ($1 \le n, m \le 100000$), the size of the array and the number of operations. The next line contains n numbers a_i, the initial state of the array ($-109 \le a_i \le 109$). The following lines contain the description of the operations. The description of each operation is as follows: i v, assign the value v to the element with index $i (0 \le i < n, -109 \le v \le 109)$.

Output

Print m+1 lines: the maximum sum of numbers on a segment before all operations and after each operation. Please note that this segment may be empty (so the sum on it will be equal to 0).

Examples

input

```
5 2
5 -4 4 3 -5
4 3
3 -1
```

output

```
8
11
7
```

input

```
4 2
-2 -1 -5 -4
1 3
3 2
```

output

```
0
3
3
```

```
#include <iostream>
using namespace std;
using ll = long long;
const int N = 100010;
int a[N];
struct SegmentTreeForTheMaximumSubSegment
{
    int l, r;
    ll mx, mlx, mrx, sum; // 最大子段和，最大前缀和， 最大后缀和，区间和
} tr[N << 2];
void pushup(int u)
```

```
{
    // 更新最大子段和
    tr[u].mx = max(tr[u << 1].mx, tr[u << 1 | 1].mx);
    tr[u].mx = max(tr[u].mx, tr[u << 1].mrx + tr[u << 1 | 1].mlx);

    // 更新最大前缀和
    tr[u].mlx = max(tr[u << 1].mlx, tr[u << 1].sum + tr[u << 1 | 1].mlx);

    // 更新最大后缀和
    tr[u].mrx = max(tr[u << 1 | 1].mrx, tr[u << 1 | 1].sum + tr[u << 1].mrx);

    // 更新区间和
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}
void build(int u, int l, int r)
{
    tr[u] = {l, r};
    if (l == r)
    {
        tr[u].mx = tr[u].mlx = tr[u].mrx = tr[u].sum = a[l];
        return;
    }
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    pushup(u);
}
void modify(int u, int k, int x)
{
    if (tr[u].l == k && tr[u].r == k)
    {
        tr[u].mx = tr[u].mlx = tr[u].mrx = tr[u].sum = x;
        return;
    }
    int mid = tr[u].l + tr[u].r >> 1;
    if (k <= mid)
        modify(u << 1, k, x);
    else
        modify(u << 1 | 1, k, x);
    pushup(u);
}
struct info
{
    ll mx, mlx, mrx, sum;
};
info compare(info a, info b)
{
    info rs;
    rs.mx = max(a.mx, max(b.mx, a.mrx + b.mlx));
    rs.mlx = max(a.mlx, a.sum + b.mlx);
    rs.mrx = max(b.mrx, b.sum + a.mrx);
    rs.sum = a.sum + b.sum;
    return rs;
}
info query(int u, int l, int r)
{
    info rs;
    if (tr[u].l >= l && tr[u].r <= r)
```

```cpp
    {
        rs.mx = tr[u].mx;
        rs.mlx = tr[u].mlx;
        rs.mrx = tr[u].mrx;
        rs.sum = tr[u].sum;
        return rs;
    }
    int mid = tr[u].l + tr[u].r >> 1;
    if (l <= mid)
    {
        rs = query(u << 1, l, r);
    }
    if (r > mid)
    {
        rs = compare(rs, query(u << 1 | 1, l, r));
    }

    return rs;
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
        cin >> a[i];
    build(1, 1, n);
    ll t = query(1, 1, n).mx;
    if (t > 0)
        cout << t << endl;
    else
        cout << 0 << endl;
    while (m--)
    {
        int op, l, r;
        cin >> l >> r;
        modify(1, l + 1, r);
        t = query(1, 1, n).mx;
        if (t > 0)
            cout << t << endl;
        else
            cout << 0 << endl;
    }
    return 0;
}

```

（二）、带懒标记的线段树

在上文的基础上，支持了区间修改操作

我们以一道例题来学习：

[一个简单的整数问题2]

给定一个长度为 N 的数列 A，以及 M 条指令，每条指令可能是以下两种之一：

1. `C l r d`，表示把 $A[l], A[l+1], \ldots, A[r]$都加上 d。
2. `Q l r`，表示询问数列中第$l \sim r$个数的和。

对于每个询问，输出一个整数表示答案。

`输入格式`

第一行两个整数 N,M。

第二行 N 个整数 A[i]。

接下来 M 行表示 M 条指令，每条指令的格式如题目描述所示。

`输出格式`

对于每个询问，输出一个整数表示答案。

每个答案占一行。

`数据范围`

$1 \leq N, M \leq 105, |d| \leq 10000, |A[i]| \leq 10^9$

`输入样例：`

```
10 5
1 2 3 4 5 6 7 8 9 10
Q 4 4
Q 1 10
Q 2 4
C 3 6 3
Q 2 4
```

`输出样例：`

```
4
55
9
15
```

分析问题：

本题维护的仍是区间和，以线段树维护，当我们进行区间修改时，可以预见的是，如果我们像之前那样一个一个点去修改，修改每一个点的时间复杂度是$O(logn)$,那么进行一次区间修改的时间复杂度就变成了$O(nlog(n))$,并且我们需要进行多次区间修改操作，这个时间复杂度是我们不可接受的，我们通过懒标记下传的方式在$O(logn)$的时间复杂度完成一次区间修改

我们的**存储方式**也要进行修改

```
struct Node
{
    int l, r; // 代表本届点维护的是区间 [l, r]的信息
    int sum, add; // 线段树要维护的信息, add作为我们的懒标记辅助完成区间修改
}tr[N * 4]; // 要开四倍空间，课上解释为何开四倍空间
```

```
// 与上文的pushup操作对应，我们需要一个将更新信息从父节点传到子节点的工具

void pushdown(int u)
{
    tr[u << 1].sum += tr[u].add * (tr[u << 1].r - tr[u << 1].l + 1);
    tr[u << 1].add += tr[u].add;
    tr[u << 1|1].sum += tr[u].add * (tr[u << 1|1].r - tr[u << 1|1].l + 1);
    tr[u << 1|1].add += tr[u].add;
    tr[u].add = 0;
}
```

注意:

这里我们强调一下$pushup$和$pushdown$的调用位置

在建树和修改的过程中，我们会从子节点向父节点更新区间信息，因此，我们的$pushup$操作要在对子节点（子树）操作之后。

而区间修改的存在，我们从上到下（从父节点到子节点）更新信息，我们的懒标记下传后，子节点再传给子节点的子节点，若在子树已经开始操作时我们的懒标记没有下传，那么我们的修改就会失败，因此，$pushdown$操作必须在修改(操作)子树之前。

由于建树过程不存在区间修改，懒标记不用下传，所以我们的$build$操作无需修改

```
void modify(int u, int l, int r, int add)
{
    if (tr[u].l >= l && tr[u].r <= r) //tr[u].sum += add * (tr[u]. r - tr[u].l + 1);
    {
        tr[u].sum += (LL)(tr[u].r - tr[u].l + 1) * add;
        tr[u].add += add;
    }
    else
    {
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, add);
        if (r > mid) modify(u << 1|1, l, r, add);
        pushup(u);
    }
}
```

```
LL query(int u, int l, int r)
{
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].sum;

    pushdown(u); // query操作也要进行懒标记下传，有可能继续递归
    int mid = tr[u].l + tr[u].r >> 1;
    LL res = 0;
    if (l <= mid) res += query(u << 1, l, r);
    if (r > mid) res += query(u << 1|1, l, r);
    return res;
}
```

完整代码

```
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int N = 100010;
typedef long long LL;
int w[N];
int n, m;
struct Node
{
    int l, r;
    LL sum, add;
}tr[N << 2];

void pushup(int u)
{
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}

void pushdown(int u)
{
    tr[u << 1].sum += tr[u].add * (tr[u << 1].r - tr[u << 1].l + 1);
    tr[u << 1].add += tr[u].add;
    tr[u << 1|1].sum += tr[u].add * (tr[u << 1|1].r - tr[u << 1|1].l + 1);
    tr[u << 1|1].add += tr[u].add;
    tr[u].add = 0;
}

void build(int u, int l, int r)
{
    tr[u] = {l, r, w[r], 0};
    if (l != r)
    {
        int mid = l + r >> 1;
        build(u << 1, l, mid), build (u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}

void modify(int u, int l, int r, int add)
{
    if (tr[u].l >= l && tr[u].r <= r) //tr[u].sum += add * (tr[u]. r - tr[u].l + 1);
    {
        tr[u].sum += (LL)(tr[u].r - tr[u].l + 1) * add;
        tr[u].add += add;
    }
    else
    {
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, add);
        if (r > mid) modify(u << 1|1, l, r, add);
        pushup(u);
    }
}

LL query(int u, int l, int r)
```

```
{
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].sum;

    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    LL res = 0;
    if (l <= mid) res += query(u << 1, l, r);
    if (r > mid) res += query(u << 1|1, l, r);
    return res;
}

int main()
{
    cin >> n >> m;
    for (int i = 1; i <= n; i ++ ) cin >> w[i];
    build (1, 1, n);
    while (m -- )
    {
        int l, r, d;
        char op[2];
        cin >> op;
        if (*op == 'C')
        {
            cin >> l >> r >> d;
            modify(1, l, r, d);
        }
        else
        {
            cin >> l >> r;
            cout << query(1, l, r) << endl;
        }
    }
}
```