# 关于数组区间问题解决算法的优化

2051498 储岱泽

**摘要**: 在我们日常工作做数据处理的过程中，往往会需要对大量的数据做加工处理，其中就不乏需要对一个存储了大量数据的数组的某些指定区间做求和、求区间平方和、求区间内最值、单点修改、单点查询等区间操作，在解决这类问题时，最传统的方式自然是直接用模拟的思想对原数组直接遍历求和以及修改，但是这样的方式在数据量小或者操作次数少的时候是高效的，然而当数据达到上万规模的时候就会发现明显的卡顿现象，大大影响了工作的效率。于是本篇论文就数组区间的一系列问题展开研究、调研、实验与分析，从前缀和数组，拓展到线段树以及树状数组这样高级的数据结构，提出了数组区间问题解决算法的优化方案。

**关键词:** 数据结构；前缀和数组；区间问题；线段树；树状数组

## The Optimization of Algorithm for Solving Range Problems

2051498 Daize Chu

**Abstract**: In the process of data processing in our daily work, we often need to process a large amount of data, among which there is no lack of range operations such as summation, range square summation, maximum value finding within the range, single point modification, single point query, etc. When solving these kinds of problems , the most traditional way is naturally to directly traverse, sum and modify the original array with the idea of simulation.This way is efficient when the amount of data is small or the number of operations is small. However, when the data reaches tens of thousands of scales, obvious "stuck" phenomenon will be found, which greatly affects the efficiency of work. Therefore, this paper conducts research, investigation, experiment and analysis on a series of problems of array interval. From prefix array to advanced data structures such as segment tree and binary indexed tree, this paper proposes an optimization scheme of range problem solving algorithm.

**Key Words**: Data structure;Prefix array; Range problem; Segment tree; Binary indexed tree

# 1 引 言

在我们日常生活统计中，常常会遇到大量的数据，并且需要处理有关这很大的一个数组某个区间的问题。比如说城市的气象研究员每一日都要记录某城市的降水量，并将其存入电脑中，某一日在研究过程中突然需要求某一段日子中降水量的总和，或者是某一段日子中降水量的最大值与最小值，又或者发现某一段时间测量仪器坏了，所测得的数据总是比真实值大 10ml，需要将这段日子测得的所有降水量都减少 10ml 再求和.....这种**对一个数组的某一个区间**进行求和、求最值、修改或是查询的操作，就叫做**区间问题**。

区间问题在生活中随处可见，不同于传统的遍历数组法对于相关问题的简单粗暴的处理，本文将提出并分析在大量数据下处理区间问题的高效方法——线段树和树状数组。

# 2 区间求和与单点修改问题优化

## 2.1 问题描述

现有下列区间问题：

给定一个长度为 n(n≤100000)的序列，序列的内容是由用户自己输入的正整数。x(x≤100000)次的修改某些位置上的数字，每次加上某一个数字，然后提出 y(y≤100000)个问题，求每段区间的和。现在要求用 C++编程，采用合理的数据结构来解决该问题。

输入格式：

第一行 1 个整数，表示序列的长度 n。

第二行 1 个整数，表示操作的次数 m。

后面依次是 m 行，分别表示单点修改或者区间求和的操作。

其中，x 表示进行单点修改，y 表示进行区间求和。

x 的格式为“x a b”表示在序列上第 a 个数加上 b。保证 1≤a≤n。

y 的格式为“y a b”表示求 a 到 b 的区间和，并将区间和的结果输出。保证 1≤a≤b≤n。

输入输出样例：

| 输入： | 输出： |
| --- | --- |
| 5 4<br>1 1 1 1 1<br>x 3 8<br>y 1 3<br>x 4 9<br>y 3 4 | 9<br>18 |

## 2.2 暴力遍历解决方案

对于此问题最容易想到的方案就是直接对数组进行暴力操作：如果要改变第 i 个元素的值，那就直接 a[i]+=b 修改；如果要求某[x,y]区间内的和，那就之间从 x 遍历到 y 全部加起来求和。

这也可以称作是用“模拟”的思想来解决这个问题，题目是怎么说的，我们就怎么做，代码也很好写，

**作者简介**:作者储岱泽，是同济大学 2021 级软件学院 2105 班的一名学生。

即：

```
while (w--) {
    cin >> choice >> a >> b;
    if (choice == 'x'){ //单点修改
            store[a - 1] += b;
            c1++;  //用于计数
    }
    else if (choice == 'y'){ //区间求和
        for (int i = a - 1; i < b; i++) {
            sum += store[i];
            c2++;  //用于计数
        }
        cout << sum << endl;
        sum = 0;
    }
```

图 1 暴力遍历解决方案的代码

这样做我们会发现，在“单点修改”的时候速度很快，平均时间复杂度仅需 O(1)，然而在进行“区间求和”的时候，其时间复杂度达到了 O(n)。

接下来我们分别以 10 个，100 个，1000 个，10000 个数据大小的数组来测试其运行的效率（均进行 20 次单点修改或者求和的操作）：

```
16
61
37
87
79
39
42
62
22
51
用于改变指定数时的查找次数：10
用于求和时的查找次数：50
运行过程中总的查找次数为：60
```

图 2 n=10 的执行效率

```
455
301
402
54
661
194
224
224
45
15
用于改变指定数时的查找次数：10
用于求和时的查找次数：417
运行过程中总的查找次数为：427
```

图 3 n=100 的执行效率

```
563
964
4960
549
5521
2175
5536
2210
838
4389
用于改变指定数时的查找次数：10
用于求和时的查找次数：4980
运行过程中总的查找次数为：4990
```

图 4 n=1000 的执行效率

```
5620
9475
54910
55009
551
21990
5531
2205
45476
4313
用于改变指定数时的查找次数：10
用于求和时的查找次数：37227
运行过程中总的查找次数为：37237
```

图 5 n=10000 的执行效率

可以发现，随着 n 的增大，求和的效率越来越低下，当 n 达到十万级的量级或者更高的时候，可能会在运行过程中出现一些明显的卡顿现象。所以，当数据量很小的时候该方案是可行的，但是当数据量很大的时候，采用该方法就不太合理了。

## 2.3 前缀和数组解决方案

在上面暴力遍历方式的基础上，我们总结出了其效率过低的主要原因是因为其区间求和比较慢，时间复杂度达到了 O(n)。那么有没有一种算法可以让求区间和快起来呢？有的。我们可以采用前缀数组的数据结构来使其求区间和的速度变快。

那么什么是前缀和数组呢？假设原数组是 a，他的前缀和数组是 b，那么必然会满足有：

b[0]=a[0]

b[1]=a[1]+b[0]=a[1]+a[0]

b[2]=a[2]+b[1]=a[2]+a[1]+a[0]

b[3]=a[3]+b[2]=a[3]+a[2]+a[1]+a[0]

.....

b[n]=a[n]+b[n-1]=a[n]+a[n-1]+...a[0]

代码如下：

```
for (int i = 0; i < n; i++) { //输入储存的数字
    cin >> a[i];
    b[i] = b[i-1] + a[i];
}
```

图 6 构建前缀和数组代码

所以前缀和数组的每一项 b[i]都表示 a 数组的前 i 项的和。因此，假如说要求数组 a 区间[x,y]内的和，只需要用 b[y]-b[x-1]即可，时间复杂度仅为 O(1)。

但是，这样做也有其弊端。如果要进行“单点修改”，则一旦修改某一点的值，其后面所有的前缀和都会发生变化，从而“牵一发而动全身”。这时候，“单点修改”的时间复杂度就为 O(n)，整体的时间效率看上去和前面的方法不相上下，但空间上的开销却比前面的方法大了 n。

代码如下：

```
while (w--) {
    cin >> choice >> a >> add;
    if (choice == 'x'){ //表示在第 a 个数上加上 b
            store[a - 1] += add;
    //接下来这一项后面的所有前缀和都需要更新
        for (int i = a - 1; i < n; i++) {
            b[i] += add;
        }
    }
    else if (choice == 'y') { //表示求[a,b]区间
```

```
中的数之和
        sum = fixSum[b] - fixSum[a - 1];
        c2++;
        ans << sum << endl;
        sum = 0;
    }
}
```

图 7 前缀和数组解决方案

接下来我们依然用同一组数据，分别以 10 个，100 个，1000 个，10000 个数据大小的数组来测试其运行的效率（均进行 20 次单点修改或者求和的操作）：

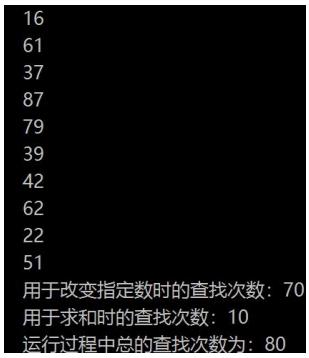


图 8 n=10 的执行效率

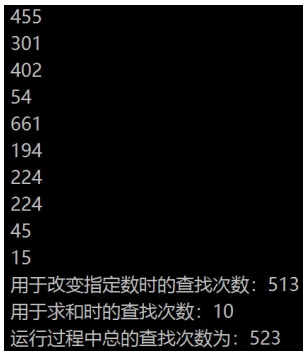


图 9 n=100 的执行效率


图 10 n=10 的执行效率


图 11 n=100 的执行效率

通过比较，发现使用暴力遍历的方式在“单点修改”上更胜一筹，而前缀和数组的解决方案在“区间求和”上更胜一筹，但综合看来都差不多。

## 2.4 使用线段树的解决方案

那么，在数据量较大的情况下，是否存在“单点修改”和“区间求和”都相对比较高效的方法呢？答案是——线段树。

线段树是一种基于树形数据结构的，用来存储各个区间信息的二叉搜索树。二叉搜索树是一种高效的可以用于检索的数据结构，其搜索效率往往能够达到 $O(\log_2 n)$。也正是如此，二叉搜索树一般被用于文件系统和数据库系统，进行高效率的排序与检索。由此可见，在有了二叉搜索树做基础的前提下，线段树这个数据结构设计出来的初衷，就是为了解决数据量巨大时的区间操作而服务的。它不仅能够高效的排序与检索，更可以高效的计算某段区间的和，或者区间、单点修改工作。

### 2.4.1 线段树的原理

线段树是一种二叉搜索树，用来存储一个区间内各个子区间的信息（各个子区间的和、最大值、最小值、乘积等等）。线段树的每一个节点都代表一个子区间，其中，根节点代表的是要维护的整个数组区间，其他节点覆盖的区间长度都为父节点区间长度的一半。

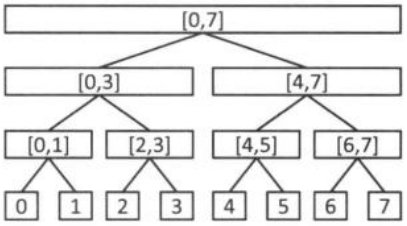


图 12 线段树数据结构示意图

假设有一个长度为 7 的数组，则可以构建成如图 12 所示的一棵线段树，每个节点标注的区间信息就是其维护的范围。这就是线段树，一种把线段构建成二叉搜索树的高级数据结构，根据二叉搜索树的时间复杂度以及性质，利用线段树来存储区间信息，其搜索与修改的时间复杂度都不会超过 $O(\log_2 n)$，可以应对较大数据的情况。

根据线段树的功能分析以及数据结构示意图，我们可以轻松的得到其每一个节点的结构：

1) l 表示该节点维护区间的左边界。

2) r 表示该节点维护区间的右边界。

3) val 表示该节点维护的区间内元素加起来的和。（根据不同的问题可以有针对性的改变）

```
//表示线段树一个节点需要存储的信息
struct Node {
    int l = 0;      //该节点维护区间的左边界
    int r = 0;      //该节点维护区间的右边界
    long long val = 0;   //该节点维护区间的和
};
```

图 13 线段树的节点结构代码

不过，在存储该线段树结构的时候，我们往往还是采用数组的形式来存储，这是因为数组通过下标来访问元素相较其链式结构的遍历来的更加快速一些。所以我们接下来需要开一个数组来存储线段树，那么这个数组应该要开多大呢？

线段树有 2 种可能的存储结果，一种是恰好可以构成一棵满二叉树，比如区间长度为 8 的情况，这个时候线段树的高度 h=4，所以说 $h=\log_2 n+1$；另外一种是比满二叉树的情况还要多出一层的平衡二叉树结构，比如区间长度为 9 的情况，这个时候线段树的高度 h=5，所以说 $h=\log_2 n+1+1$。由此可见，为了能够使得数组存下左右的节点，至少应该能够存储下一棵高度为

$h=\log_2 n+1+1$ 的满二叉树。根据满二叉树的性质，每一层存储的元素个数是 $2^{n-1}$ 个。所以根据等比数列求和：

$$S=\frac{2^0\bullet(1-2^{\log_2 N+1+1})}{1-2}=2^{\log_2 N+2}-1=4N-1$$

因此可知，存储线段树的数组至少应该开 4 倍区间的大小，所以我们 `Node tree[N * 4];`，由此可见，线段树的空间复杂度是 O(4n)，远远大于前面提到的两种算法，是一种“以空间换时间”的算法。

在该数组中，也像是一般的二叉树一样，对于下标为 x 的节点，它的左孩子节点下标为 2x，它的右孩子节点下标为 2x+1。为了使得后面的代码可读性更高，在程序的开头可以先引进两个内联函数，用来计算某个节点的左孩子和右孩子：

```
inline int ls(int p) {//p 的左儿子
    return (p << 1);
}
inline int rs(int p) {//p 的右儿子
    return (p << 1 | 1);
}
```

图 14 求左右孩子的内联函数

### 2.4.2 线段树建树

在 build 函数中有三个参数：
1)i 代表的是当前节点在 tree 数组中的编号。
2)l 表示当前节点维护的区间的左边界。
3)r 表示当前节点维护的区间的右边界。

线段树的建树方式是采用递归的方式，先从根节点开始对左半棵子树无限二分递归，直到二分到叶子节点，然后再输入这个叶子节点的值。然后逐层向上更新每一个父节点的值（父节点的 val 值等于每一个孩子节点 val 值之和，这边用到了自定义的内联函数 push_up()）。接下来对右半棵子树做同样的操作。

```
inline void push_up(int p) {//自底向上合并
  tree[p].val=(tree[ls(p)].val+tree[rs(p)].val);
}
void build(int i, int l, int r) {
    tree[i].l = l;
    tree[i].r = r;
    if (l == r) {//叶子节点
        cin >> tree[i].val;
        return;
    }
    int mid = (l + r) >> 1;
    build(ls(i), l, mid);//递归构建左子数
    build(rs(i), mid + 1, r);//递归构建右子树
    push_up(i);
}
```

图 15 线段数的建树代码

### 2.4.3 线段树单点修改

下面是线段树用于实现单点修改的函数 change_point，其中一共有五个参数：
1)k 代表的是当前节点在 tree 数组中的编号。
2)l 表示当前节点维护的区间的左边界。
3)r 表示当前节点维护的区间的右边界。
4)x 表示的是需要修改的节点在原序列中的位置。
5)v 表示的是该节点需要增加的值。

线段树在进行单点修改的时候依旧是利用了递归的思想，无限二分递归下去寻找需要修改的那个节点，如果找到了，就给这个节点加上 v，然后返回，并且往回走，将之前走过的节点的 val 都重新更新一遍。

```
void change_point(int k,int l,int r,int x,int v){
    c1++;//用于计数，统计单点修改时函数执行次数
    if (tree[k].l == tree[k].r&&tree[k].l==x){
        tree[k].val += v;
        return;
    }
    int mid = (l + r) >> 1;
    if (x <= tree[ls(k)].r)
        change_point(ls(k), l,mid,x,v);
    else
        change_point(rs(k), mid+1, r, x, v);
  tree[k].val=tree[ls(k)].val+tree[rs(k)].val;
    return;
}
```

图 16 线段数的单点修改代码

### 2.4.4 线段树区间查询以及求和

在 search 函数中一共有 5 个参数：
1)k 代表的是当前节点在 tree 数组中的编号。
2)l 表示当前节点维护的区间的左边界。
3)r 表示当前节点维护的区间的右边界。
4)x 表示的是需要求和的区间的左边界。
5)v 表示的是需要求和的区间的右边界。

求和的时候，如果当前遍历到的节点所维护的区间恰好包含在了我要查询的区间里面，那就直接返回该节点的值，也就是该区间的和，将其加在结果之中，如果不是，那就返回一个不影响结果的值“0”加在结果后面。

```
long long search(int k, int l, int r, int x, int y){
    c2++;//用于计数，统计区间求和时函数执行次数
//如果这个区间被完全包括在目标区间里面，直接返回这个区间的值
    if (tree[k].l >= x && tree[k].r <= y)
        return tree[k].val;
```

```
//如果这个区间和目标区间毫不相干，返回 0
    if (tree[k].r<x || tree[k].l>y)
        return 0;
    int s = 0;
    int mid = (l + r) >> 1;
    if (tree[ls(k)].r >= x)
        s += search(ls(k), l, mid,x,y);
    if (tree[rs(k)].l <= y)
        s += search(rs(k),mid+1,r,x,y);
    return s;
}
```

图 17 线段数的区间求和代码

### 2.4.5 运行结果

这样一来就把运用线段树来进行“单点修改”与“区间求和”的代码分析清楚了，接下来运用和前面测试暴力遍历以及前缀和数组的解决方案一样的四组数组来测试在 n=10,n=100,n=1000 以及 n=10000 下，线段树处理该类问题的运算效率：

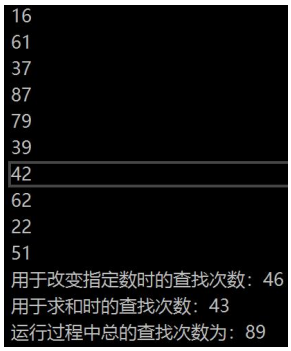


图 18 n=10 的执行效率

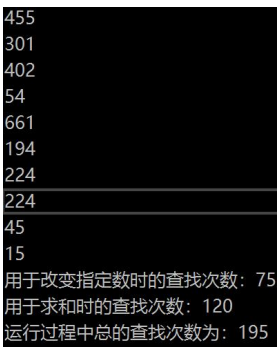


图 19 n=100 的执行效率


图 20 n=1000 的执行效率


图 21 n=10000 的执行效率

通过和上面的数组暴力遍历法以及前缀和数组法相比较，可以发现利用线段树来处理这类问题，在 n 比较小的时候优势并没有很明显，但是当 n 很大的时候，在效率上可以有明显的提升，并且 n 越大，这种效率的优势就越明显。

### 2.5 使用树状数组解决方案

然而在处理这一类问题上，基于线段树的基础我们还可以做进一步的优化。

假设现在有一个长度为 16 的区间，其元素存在数组 a 内，a[16]={1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6}，则其在线段树上面的存储方式大致可以用下面的表格来模拟。即：最顶上的 tree[1]表示着整个区间[1,16]内所有元素的和，下面左孩子和右孩子平均分为两个区间，左孩子表示区间[1,8]的和，是 36；右孩子表示区间[9,16]的和，是 40。接下来再在各自的左孩子与右孩子下进行区间的平分。这样子，假如说我要求[1,7]内的和，就不需要从 1 一直遍历到 7，只需要用[1,4]+[5,6]+[7]即可，即：10+11+7=28。

表 1 线段树存储示意

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 76 | | | | | | | | | | | | | | | |
| 36 | | | | | | | | 40 | | | | | | | |
| 10 | | | | 26 | | | | 22 | | | | 18 | | | |
| 3 | | 7 | | 11 | | 15 | | 19 | | 3 | | 7 | | 11 | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 1 | 2 | 3 | 4 | 5 | 6 |

那么，如果我们要求的是[1,3]的和呢？我们只需要用[1,2]+[3,3]，即：3+3=6。如果我们要求的是[1,4]的和，我们只需要[1,4]，即：直接就能得到 10。

通过上面的例子，我们发现[3,4]的这一节存储在求前 n 项和这个问题上貌似一直是多余的，而如果我们要求[n,m]区间的和可以直接用前 m 项的和减去前 n-1 项的和，所以无论是求[3,4]还是求[1,3]、[1,4]，[3,4]这一节存储“3 和 4 的和”都可以不派上用场，即便去掉了也可以。事实上，通过观察我们发现，类似于这一个区间的性质的区间还有很多，比如[7,8]，如果要求[1,7]可以用[1,4]+[5,6]+[7,7]，求[1,8]可以直接看[1,8]，用不上[7,8]这个区间.....每一行的第偶数个存储单位都是多余的，即下表中用阴影标注的单元格都是可有可无的，去掉也不影响计算：

表 2 阴影部分单元格消失也不影响前 n 项和的计算

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 76 | | | | | | | | | | | | | | | |
| 36 | | | | | | | | 40 | | | | | | | |
| 10 | | | | 26 | | | | 22 | | | | 18 | | | |
| 3 | | 7 | | 11 | | 15 | | 19 | | 3 | | 7 | | 11 | |
| 1 | | 3 | | 5 | | 7 | | 9 | | 1 | | 3 | | 5 | |

将这些打上阴影的部分去掉之后，仔细数一下剩下的数字，我们发现恰巧还剩下 16 个数字，可以装进原来的数组之中：

表 3 剩下的数正好装进一个和原来大小一样的数组中

| 76 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 36 | | | | | | | | | | | | | | | |
| 10 | | | | | | | | 22 | | | | | | | |
| 3 | | | | 11 | | | | 19 | | | | 7 | | | |
| 1 | | 3 | | 5 | | 7 | | 9 | | 1 | | 3 | | 5 | |
| 1 | 3 | 3 | 10 | 5 | 11 | 7 | 36 | 9 | 19 | 1 | 22 | 3 | 7 | 5 | 76 |

这和原来的老方法“线段树”需要额外开一个 4N 大小的数组才能存的下相比，现在只需要开一个大小为 N 的数组，在空间复杂度上降低了不少，节约了很多空间。

而这个存储了剩下的这 16 个数的数组，就是一棵树状数组。和线段树类似的是，树状数组中的每一个元素也表示了原数组的某个区间的和。我们求和时，只需要找到对应的区间，将这些区间相加即可找到答案。修改某个数据时，我们也只需要向上找到包含它的区间进行修改即可。

知道了他们是怎么存储的，那么这些存储在树状数组中的元素下标又有什么规律呢？

表 4 每行的一个单元表示的区间大小恰好是二进制表达最低位

| 长度 | 原树状数组表 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 76 | | | | | | | | | | | | | | | |
| 8 | 36 | | | | | | | | 40 | | | | | | | |
| 4 | 10 | | | | 11 | | | | 22 | | | | 7 | | | |
| 2 | 3 | | 3 | | 11 | | 7 | | 19 | | 1 | | 7 | | 5 | |
| 1 | 1 | 1 | 3 | 3 | 5 | 11 | 7 | 36 | 9 | 19 | 1 | 22 | 3 | 7 | 5 | 76 |

表 5 下标二进制表达的最低位的十进制大小与区间长度比较

| 十进制数值 | 二进制形式 | 最低非 0 位大小 | 该位涵盖区间大小 |
|---|---|---|---|
| 1 | 0001 | $(1)_{10}$ | 1 |
| 2 | 0010 | $(2)_{10}$ | 2 |
| 3 | 0011 | $(1)_{10}$ | 1 |
| 4 | 0100 | $(4)_{10}$ | 4 |
| 5 | 0101 | $(1)_{10}$ | 1 |
| 6 | 0110 | $(2)_{10}$ | 2 |
| 7 | 0111 | $(1)_{10}$ | 1 |
| 8 | 1000 | $(8)_{10}$ | 8 |

通过上面两张表我们可以发现，数组下标的二进制表达形式的最低位的一个“1”所表示的十进制数的大小，恰巧和该位所涵盖的区间的大小相等。我们姑且将其称作 lowbit。

下面是求 lowbit 的函数：

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

图 22 lowbit 函数的代码

然后我们每次给下标为 i 的 a[i]加上 v，都需要继续向上寻找包含它的所有区间，给这些区间都加上 v。举个例子，现在我要给 a[2]加上 3，那么 a[2]就从原来的 2 变成了 5。因此下面标成深灰色的元素都发生了变化：

表 6 给 a[2]加 3 之后各个区间的变化

| 79 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 39 | | | | | | | | 22 | | | | | | | |
| 13 | | | | 11 | | | | 19 | | | | 7 | | | |
| 5 | | | | | | | | | | | | | | | |
| 1 | 5 | 3 | 13 | 5 | 11 | 7 | 39 | 9 | 19 | 1 | 22 | 3 | 7 | 5 | 79 |

我们发现，树状数组的第 2 项，第 4 项，第 8 项和第 16 项都加上了 3，而 lowbit(2)=4-2，lowbit(4)=8-4,lowbit(8)=16-8。所以我们发现，所有要改变的项的下标等于前一个改变的项的下标 lowbit 值加上自身，一直到 n 为止，因此我们得到下面树状数组用来“单点修改”的函数。

```
void add(int p, int x) {
    while (p <= n) {
      c1++;//用于计数，统计单点修改时的循环次数
        a[p] += x;
        p += lowbit(p);
    }
}
```

图 23 树状数组“单点修改”函数的代码

那么区间求和呢？在树状数组中如果想要求前 n 项的和，则有点像是和“单点修改”是倒着来的，下面看一下区间求和的函数：

```
int ask(int p) {
    int ans = 0;
    while (p > 0) {
      c2++;//用于计数，统计区间求和循环执行次数
        ans += a[p];
        p -= lowbit(p);
    }
    return ans;
}
```

图 24 树状数组“区间求和”函数的代码

以上便是树状数组关于“单点修改”以及“区间求和”的全部代码，看上去比起线段树是不是写起来简洁了很多呢？

接下来让我们用与上面三组测试相同的四组测试分别测试在 n=10,n=100,n=1000,n=10000 下树状数组的运行效率：

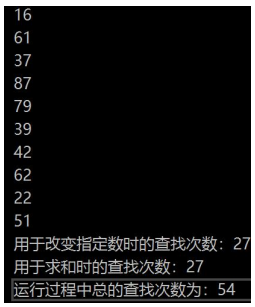

图 25 n=10 的执行效率

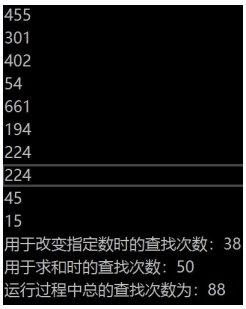

图 26 n=100 的执行效率

图 27 n=1000 的执行效率

图 28 n=10000 的执行效率

太神奇了！竟然比线段树的效率还高出了好多！由此可见，对于此类问题树状数组的性能比线段树更优，并且所需要的额外存储空间也更少，适合用来处理数据量庞大的区间问题。

# 3 结论

通过实验用同样的一组数据测试上面四种不同的解决方案，我们总结出了下面这这张表：

表 7 各种方法的时间与空间复杂度

| | 单点修改 | | 区间求和 | |
|---|---|---|---|---|
| 复杂度 | 时间 | 空间 | 时间 | 空间 |
| 暴力遍历 | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ |
| 前缀和数组 | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ |
| 线段树 | $O(\log_2 n)$ | $O(4n)$ | $O(\log_2 n)$ | $O(4n)$ |
| 树状数组 | $O(\log_2 n)$ | $O(n)$ | $O(\log_2 n)$ | $O(n)$ |

于是，对于上述四个方法适用的场合，我们可以做以下总结：

1) 如果在接下来的工作当中主要以单点修改为主，那么可以直接在数组中通过下标访问该元素进行修改；如果以区间求和为主，那么可以用前缀和数组的方式进行求区间的和。

2) 如果在接下来的工作中处理的数据量不是很大，那么这四种方案的效率差别不大。

3) 如果工作中要处理的数据量很大，是成千上万的级别，那么线段树和树状数组的时间复杂度仅为 $O(\log_2 n)$，远远高于其他两种方式。然而在实验中我们也发现，树状数组的空间复杂度 $O(n)$ 也远远低于线段树 $O(4n)$，且代码更加简洁运行效率更高，所以还是首选树状数组。

4) 然而线段树的功能相对来说比树状数组更多，如果遇到区间修改的问题，即对一整个区间上的数进行操作，还是线段树更方便些；还有如果需要求的是区间的最大值和最小值，也还是得用线段树进行处理.....综上所述，线段树的功能比树状数组更多，但是就“单点修改”和“区间求和”这两个问题上来说，树状数组更适合也更简洁，杀鸡不必用牛刀。