树状数组

求前缀和

我们的目标是——

- 树状数组的存储结构
- 树状数组的计算方法
- 树状数组支持的操作
  - 单点修改，区间查询（当区间宽度为1时即单点查询）

  这是树状数组支持的操作，下面的两种是其变形，但本质还是单点修改，区间查询
  - 区间修改，单点查询
  - 区间修改，区间查询

【例】求连续和

【问题描述】

求长度为n的数组A在（1<=a<b<=n <10^5）之间的总和。

```
int c = 0;
for (int i = a; i <= b; ++i) {
    c += A[i];
}
```

时间复杂度为O(n)，最大执行次数为10^5

【例】求连续和

如果题目改成求q（q <= 10^5）次连续和呢？

时间复杂度O(q*n)，1s内极限数据会TLE

优化代码：

```
int C[N];
for (int i = 1; i <= n; ++i) { //求前缀和
      C[i] = C[i-1] + A[i];
}
printf("%d\n", C[b] - C[a-1]);
```

单次查询时间复杂度为O(1)，总时间复杂度O(q+n)

【例】求连续和

如果题目增加个操作：需要频繁修改A[i]（0 < i <= n）

此时前缀和C[i]…C[n]都将被修改。

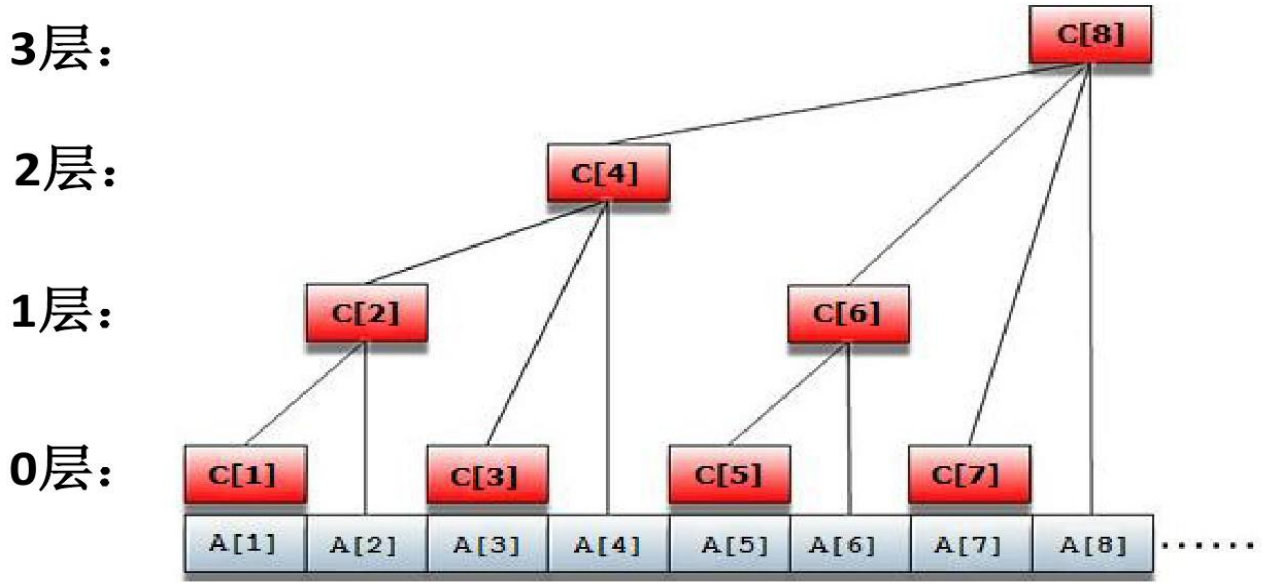
如果依次对C[i]…C[n]进行修改，时间复杂度为O(n)。则该题总时间复杂度为O(q*n)，依旧TLE。

因此，该问题的重心转移到如何能够高效地维护前缀和。

树状数组

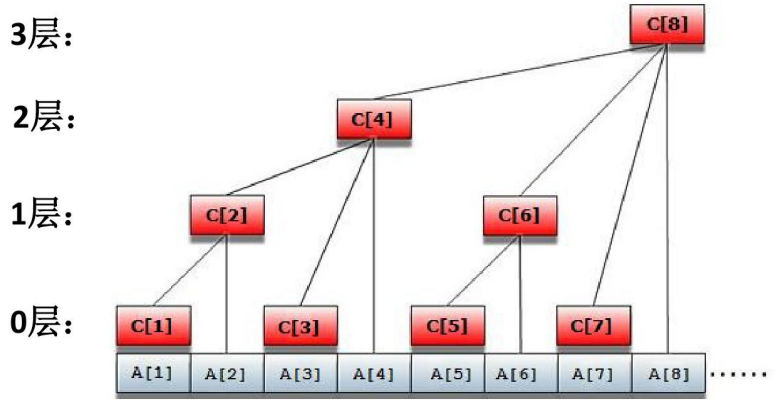
定义

- 树状数组是一个查询和修改复杂度都为log(n)的数据结构。主要用于查询任意区间的连续元素和，但是每次只能修改一个元素的值；即树状数组支持的操作：单点修改，区间查询（当区间长度为1时，即单点查询）
- 树状数组逻辑上是一棵树，但实际上只是一个数组。



找规律

- A数组保存原数据，C数组为前缀和数组。
- C数组的计算规律如下：
  - C1 = A1
  - C2 = A1 + A2
  - C3 = A3
  - C4 = A1 + A2 + A3 + A4
  - C5 = A5
  - C6 = A5 + A6
  - C7 = A7
  - C8 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8



- 设C的某个元素下标为x，则这个结点（前缀和）的管辖区间是2^k个元素（其中k为x的二进制数的末尾0的个数），且该区间的最后一个元素为Ax
- Cx的双亲结点下标y就等于x的二进制数在最后一个1的位置上加1后得到的值。

二进制基础

正数的8位二进制编码

十进制	原码	反码	补码
1	0000 0001	0000 0001	0000 0001
2	0000 0010	0000 0010	0000 0010
3	0000 0011	0000 0011	0000 0011
4	0000 0100	0000 0100	0000 0100
7	0000 0111	0000 0111	0000 0111
8	0000 1000	0000 1000	0000 1000
12	0000 1100	0000 1100	0000 1100
14	0000 1110	0000 1110	0000 1110
15	0000 1111	0000 1111	0000 1111

负数的8位二进制编码

十进制	原码	反码	补码
-1	1000 0001	1111 1110	1111 1111
-2	1000 0010	1111 1101	1111 1110
-3	1000 0011	1111 1100	1111 1101
-4	1000 0100	1111 1011	1111 1100
-7	1000 0111	1111 1000	1111 1001
-8	1000 1000	1111 0111	1111 1000
-12	1000 1100	1111 0011	1111 0100
-14	1000 1110	1111 0001	1111 0010
-15	1000 1111	1111 0000	1111 0001

求C[x]的管辖范围

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

$x = (7)_{10} = (0000\ 0111)_2$

$x = (-7)_{10} = (1111\ 1001)_2$

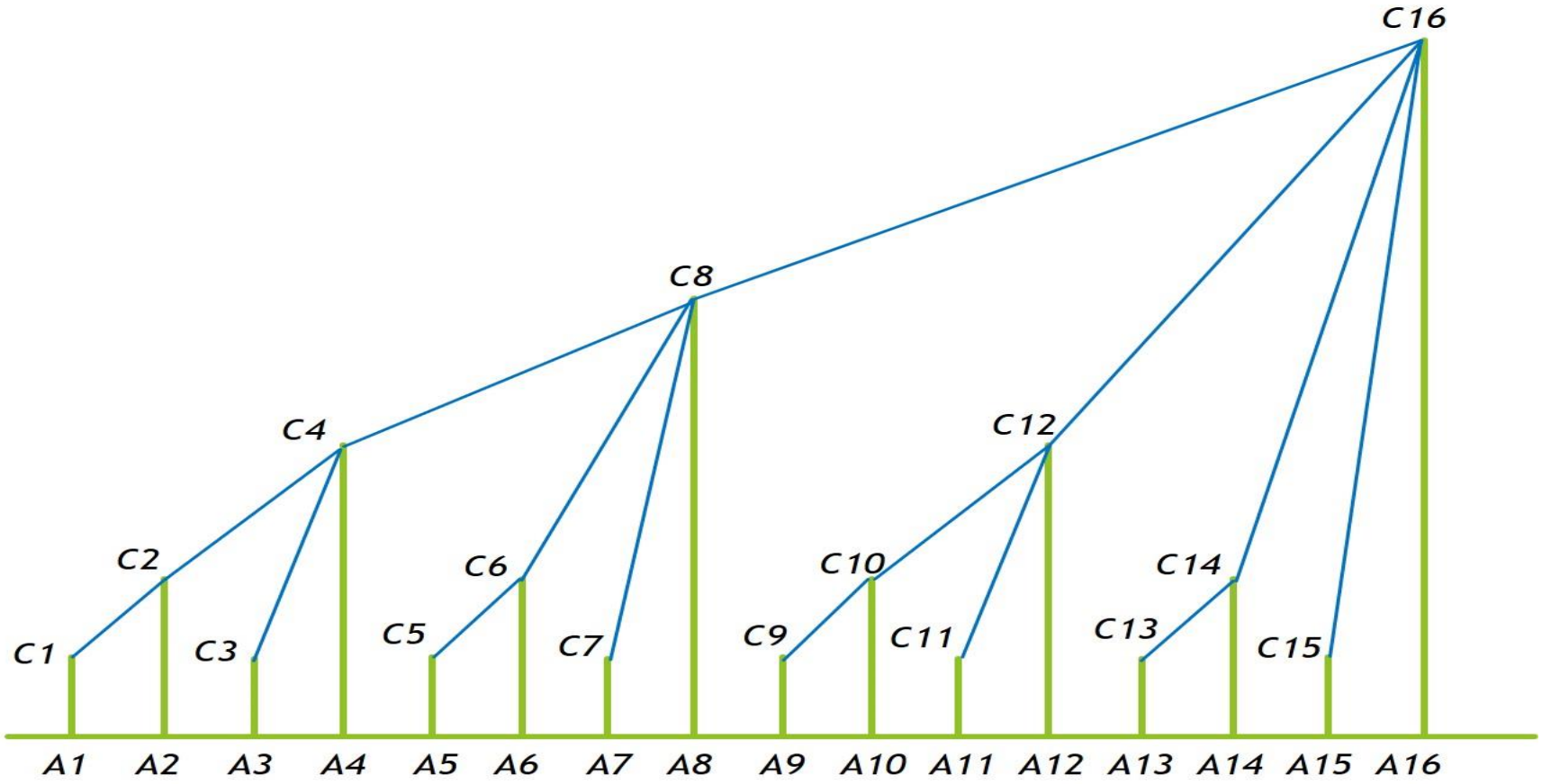
$lowbit(7) = (0000\ 0001)_2 = (1)_{10}$

$x = (12)_{10} = (0000\ 1100)_2$

$x = (-12)_{10} = (1111\ 0100)_2$

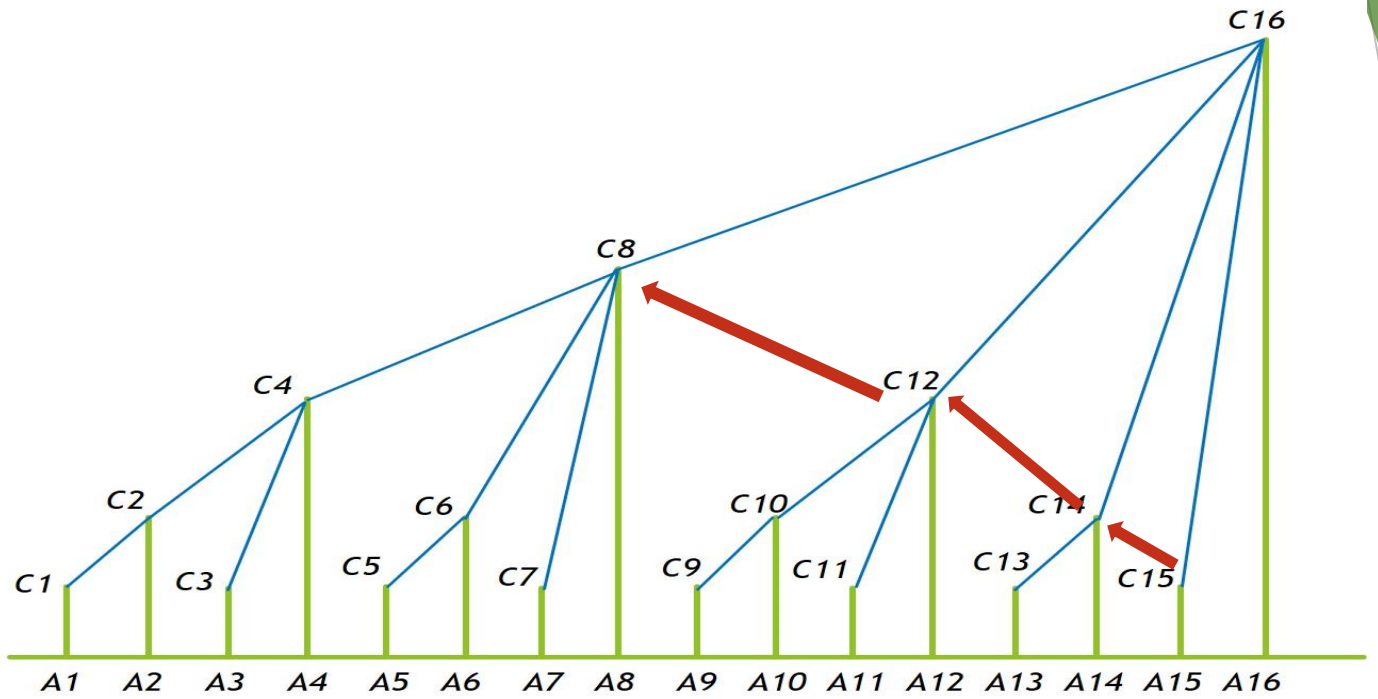
$lowbit(12) = (0000\ 0100)_2 = (4)_{10}$

区间查询



前缀和sumn[15] = A[1] + … + A[15] = C15 + C14 + C12 + C8

区间查询




```
int getsum(int x) {
	int s = 0;
	while (x > 0)	{
		s += C[x];
		x -= lowbit(x);
	}
	return s;
}

printf("%d\n", getsum(b) - getsum(a-1));
```

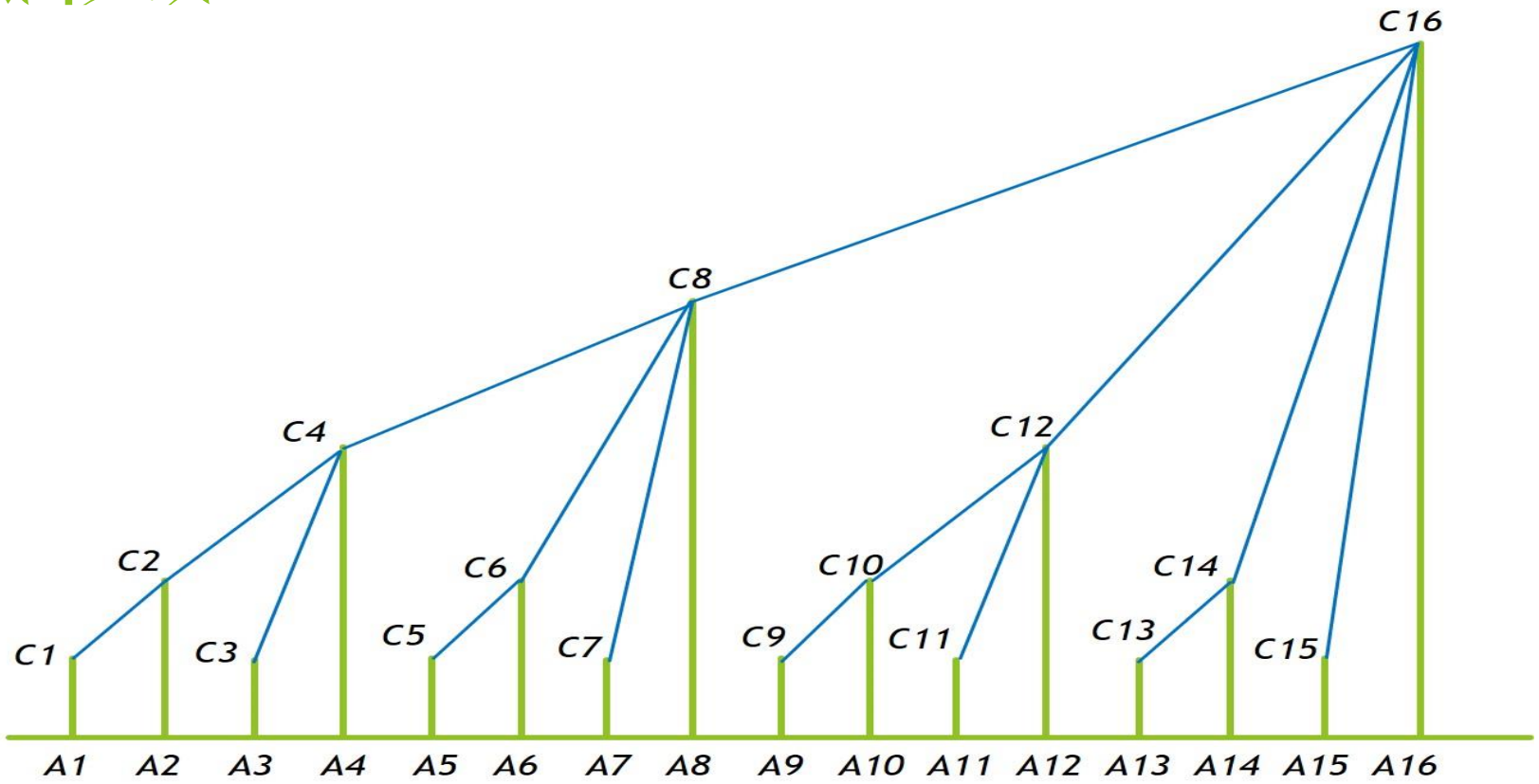
单次查询的时间复杂度为

$O(\log_2(n))$

总时间复杂度为

$O(q*\log_2(n))$

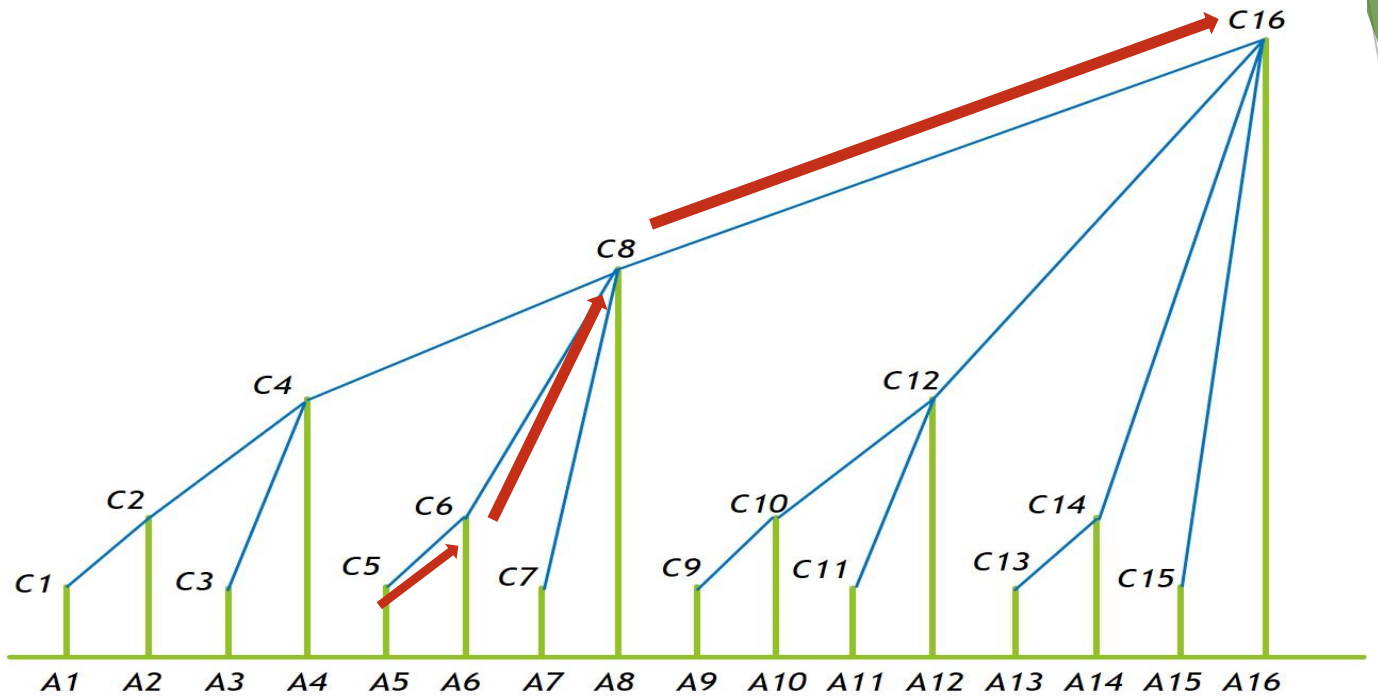
单点修改



如果要修改A[5]，则需要更新哪些值？

C5, C6, C8, C16

单点修改




```
int add(int n, int x, int key) {// A[i]增加key
    while (x <= n) {
        C[x] += key;
        x += lowbit(x); // 向更高一层更新
    }
}
```

单次修改的时间复杂度为

$O(\log_2(n))$

总时间复杂度为

$O(q*\log_2(n))$

区间修改，单点查询

【问题描述】
N个气球排成一排，从左到右依次编号为1,2,3....N。每次给定2个整数a和b(a<=b)，lele便为骑上他的“小飞鸽”牌电动车从气球a开始到气球b依次给每个气球涂一次颜色。但是N次以后lele已经忘记了第i个气球已经涂过几次颜色了，你能帮他算出每个气球被涂过几次颜色吗？
【输入格式】
每个测试实例第一行为一个整数N(N<=100000)。
接下来的N行，每行包括2个整数a b(1<=a<=b<=N)。当N=0，输入结束。
【输出格式】
每个测试实例输出一行，包括N个整数，第i个数代表第i个气球总共被涂色的次数。

【样例输入】

```
3
1 1
2 2
3 3
3
1 1
1 2
1 3
0
```

【样例输出】

```
1 1 1
3 2 1
```

区间修改，单点查询

- 区间修改的特点——　区间中的每个值变化量都一样
- 修改区间[l, r]的值，不变的是什么？

  区间中的相邻两个元素的差值
  另**B[i] = A[i] – A[i-1]**，特别的**B[1] = A[1]**，
  用**C**数组表示**B[i]**的前缀和。
- 修改区间[l, r]的值后，变化的还有什么？

  B[l]和**B[r+1]**
- 修改区间[l, r]的值，可以转化成什么？

  ```
  B[l] += key;
  B[r+1] -= key;
  ```
- 单点查询A[i] = ？　**A[i] = B[1] + B[2] + … + B[i]**

区间修改，单点查询

【问题描述】
N个气球排成一排，从左到右依次编号为1,2,3....N。每次给定2个整数a和b(a<=b)，lele便为骑上他的“小飞鸽”牌电动车从气球a开始到气球b依次给每个气球涂一次颜色。但是N次以后lele已经忘记了第i个气球已经涂过几次颜色了，你能帮他算出每个气球被涂过几次颜色吗？
【输入格式】
每个测试实例第一行为一个整数N(N<=100000)。
接下来的N行，每行包括2个整数a b(1<=a<=b<=N)。当N=0，输入结束。
【输出格式】
每个测试实例输出一行，包括N个整数，第i个数代表第i个气球总共被涂色的次数。

【样例输入】

```
3
1 1
2 2
3 3
3
1 1
1 2
1 3
0
```

【样例输出】

```
1 1 1
3 2 1
```

区间修改，单点查询

- 主要代码——

```
memset(c, 0, sizeof(c));
scanf("%d", &n);
int l, r;
for (int i = 0; i < n; ++i) {
    scanf("%d%d", &l, &r);
    add(l, 1);
    add(r+1, -1);
}
for (int i = 1; i <= n; ++i) {
    printf("%d ", getsum(i));
}
```

区间修改，区间查询

【问题描述】
给你N个整数A[1], A[2], ... , A[N]。你需要处理两类问题：
“C a b c”表示给A[a], A[a+1], ... , A[b]之间的每个数都加上c(-10000≤c≤10000)。
“Q a b”求A[a], A[a+1], ... , A[b]之间数字的总和；
【输入格式】
输入的第一行包含两个整数N和Q(1≤N,Q≤100000)；
第二行包含N个整数Ai(-10^9≤Ai≤10^9)
接下来Q行，表示Q个问题，形式如题；
【输出格式】
输出要求计算出的区间总和，每行一个。

【样例输入】	【样例输出】
10 5 1 2 3 4 5 6 7	4
8 9 10	55
Q 4 4	9
Q 1 10	15
Q 2 4	
C 3 6 3	
Q 2 4	

区间修改，区间查询

- 区间修改没问题了，区间查询怎么解决？

  区间查询即区间内单点查询结果的和

  仍然沿用c数组，考虑A数组[1,x]区间和的计算。b[1]被累加了x次，b[2]被累加了x-1次，...，b[x]被累加了1次。因此得到

  $$\sum a[i]=\sum\{b[i]*(x-i+1)\}=\sum\{b[i]*(x+1)-b[i]*i\}=(x+1)*\sum b[i]-\sum(b[i]*i)$$

  所以我们再用树状数组维护一个数组b2[i]=b[i]*i，即可完成任务。
  用树状数组C1维护b[i]的前缀和，C2维护b2[i]的前缀和。

区间修改，区间查询

- 主要代码——

```
for (int i = 1; i <= n; ++i) { // 初始化处理差分
    scanf(“%lld”, &num[i]);
    add(i,num[i]-num[i-1]);
}
void add(int x,int y){ //给差分数组中的位置x加上y
        int i = x;
        while (i <= n) {
                c1[i] += y;
                c2[i] += (long long)x * y;
                i += lowbit(i);
        }
}
long long getsum(int x){//查询前x项的和
    long long ans = 0;
    int i = x;
    while (i > 0) {
        ans += (x+1)*c1[i] - c2[i];
        i -= lowbit(i);
    }
    return ans;
}
```

getsum(b) – getsum(a-1)为最后的答案

区间修改，区间查询

- 另一种思路：
  - 首先，看更新操作Update(s,t,d)把区间A[s]...A[t]都增加d，我们沿用数组b[i]，表示的意义修改一下，即表示A[i]...A[n]的共同增量，n是数组的大小。那么update操作可以转化为：
  - 1）令b[s]=b[s]+d，表示将A[s]...A[n]同时增加d，但这样A[t+1]...A[n]就多加了d
  - 2）再令b[t+1]=b[t+1]-d，表示将A[t+1]...A[n]同时减d；
  - 然后来看查询操作Ask(s,t)，求A[s]...A[t]的区间和，转化为求前缀和，设sum[i]= A[1]+...+A[i]，则A[s]+...+A[t]=sum[t]-sum[s-1]，那么前缀和sum[x]又如何求呢？它由两部分组成，一是数组的原始和，二是该区间内的累计增量和，把数组A的原始值保存在数组org中，并且b[i]对sum[x]的贡献值为b[i]*(x+1-i)，那么
  - sum[x]=org[1]+...+org[x]+b[1]*x+b[2]*(x-1)+b[3]*(x-2)+...+b[x]*1
  - =org[1]+...+org[x]+segma(b[i]*(x+1-i))
  - =segma(org[i])+(x+1)*segma(b[i])-segma(b[i]*i)，1<=i<=x
  - 这其实就是三个数组org[i], b[i]和b[i]*i的前缀和，org[i]的前缀和保持不变，事先就可以求出来，b[i]和b[i]*i的前缀和是不断变化的，可以用两个树状数组来维护。

区间修改，区间查询

- 主要代码——

```
// 输入数据，求普通的前缀和备用
for(i=1;i<=n;i++)scanf("%d",&A[i]);
for(i=1;i<=n;i++)sum[i]=sum[i-1]+A[i];

while(q--) {
	while(ch=getchar())if(ch=='C'||ch=='Q')break;
	if(ch=='Q') {
		scanf("%d%d",&s,&t);
		Ans=sum[t]-sum[s-1];
		Ans+=(t+1)*Getsum(c1,t)-Getsum(c2,t);
		Ans-=s*Getsum(c1,s-1)-Getsum(c2,s-1);
		printf("%lld\n",Ans);
	} else {
		scanf("%d%d%d",&s,&t,&d);
		//把差分数组c[i](s<=i<=t)加d，策略是
		//先把[s,n]内的增量加d，再把[t+1,n]的增量减d
		Add(s,d);
		Add(t+1,-d);
	}
}
```

二维树状数组

- 设二维数组为：

```
a[][]={ {a11,a12,a13,a14,a15,a16,a17,a18},
        {a21,a22,a23,a24,a25,a26,a27,a28},
        {a31,a32,a33,a34,a35,a36,a37,a38},
        {a41,a42,a43,a44,a45,a46,a47,a48}};
```

那么C[1][1] = a11,C[1][2] = a11 + a12;
如此当C[1][i]...C[1][j]时跟一维的树状数组是没有什么区别的
那么C[2][1] = a11 + a21,C[2][2] = a11 + a12 + a21 + a21,
如此可以发现
其实C[2][i].....C[2][j],就是C[1][],C[2][],单独的两个一维树状数组同一位置的值合并在一起
而C[3][1] = a31,C[3][2] = a31 + a32 ......
而C[4][1] = a11 + a21 + a31 + a41,C[4][2] = a11 + a12 + a21 + a22 + a31 + a32 + a41 + a42

所以二维数组的规律就是,不管是横坐标还是纵坐标,将他们单独拿出来,他们都符合x += lowbit(x),属于它的父亲节点.

二维树状数组

- 更新部分代码如下——

```
/*如果我改变了A[x][y]这个点,那么接下来C[x][y += lowbit(y)]当做一维数组的话都是要改变一个key的；接着我们的行坐标也是要改变的，
C[x += lowbit(x)][y]也是要改变的，因为他们都包含了A[x][y]这个点*/

void add(int x, int y, int key) {
for(int i = x; i <= n; i += lowbit(i)) {
        for(int j = y; j <= n; j += lowbit(j)) {
            C[i][j] += key;
        }
    }
}
```