# 2.3 树状数组

## 原理 1 一维树状数组

有一个包含 $n$ 个数的数列 2,7,1,12,5,9…，请计算前 $i$ 个数的和值，即前缀和 sum[$i$]=$a$[1]+$a$[2]+…+$a$[$i$]（$i$=1,2,…,$n$）。该怎么计算呢？一个一个加起来怎么样？

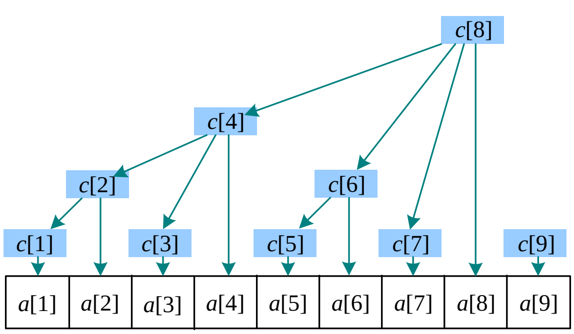
```
sum=0;
for(int k=1;k<=i;k++)
    sum+=a[k];
```

若用这种办法，则计算前 $n$ 个数的和值需要 $O(n)$时间。而且若对 $a$[$i$]进行修改，则对 sum[$i$],sum[$i$+1],…,sum[$n$]都需要修改，在最坏的情况下需要 $O(n)$时间。当 $n$ 特别大时效率很低。

树状数组可以高效地计算数列的前缀和，其查询前缀和与点更新（修改）操作都可以在 $O(\log n)$时间内完成，那么树状数组是怎么巧妙实现这些的呢？

### 1. 树状数组的由来

树状数组引入了分级管理制度且设置了一个管理小组，管理小组中的每个成员都管理一个或多个连续的元素。例如，在数列中有 9 个元素，分别用 $a$[1],$a$[2],…,$a$[9]存储，还设置了一个管理小组 $c$[]。



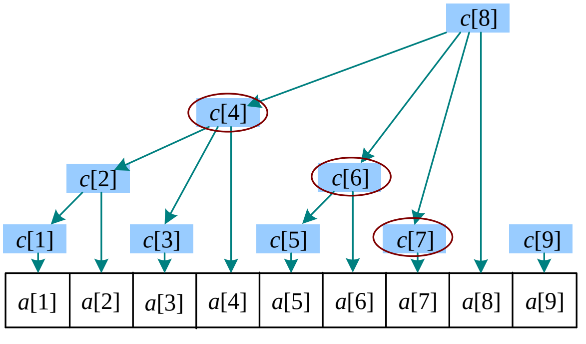
管理小组的每个成员都存储其所有子节点的和。

- *c*[1]：存储 *a*[1]的值。
- *c*[2]：存储 *c*[1]、*a*[2]的和值，相当于存储 *a*[1]、*a*[2]的和值。
- *c*[3]：存储 *a*[3]的值。
- *c*[4]：存储 *c*[2]、*c*[3]、*a*[4]的和值，相当于存储 *a*[1]、*a*[2]、*a*[3]、*a*[4]的和值。
- *c*[5]：存储 *a*[5]的值。
- *c*[6]：存储 *c*[5]、*a*[6]的和值，相当于存储 *a*[5]、*a*[6]的和值。
- *c*[7]：存储 *a*[7]的值。
- *c*[8]：存储 *c*[4]、*c*[6]、*c*[7]、*a*[8]的和值，相当于存储 *a*[1]～*a*[8]的和值。
- *c*[9]：存储 *a*[9]的值。

从上图可以看出，这个管理数组 *c*[]是树状的，因此叫作树状数组。怎么利用树状数组求前缀和及点更新呢？

**1）查询前缀和**

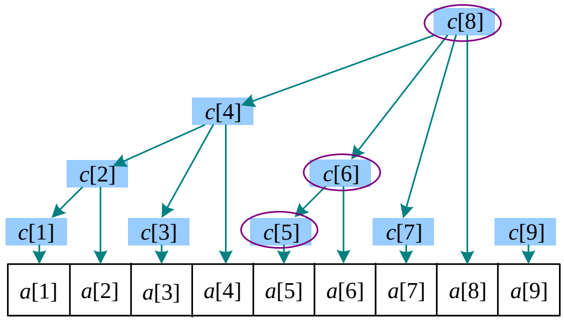
若想知道 sum[7]，则只需 *c*[7]加上左侧所有子树的根即可，即 sum[7]=*c*[4]+*c*[6]+*c*[7]。




- sum[4]：左侧没有子树，直接找 *c*[4]即可，sum[4]=*c*[4]。
- sum[5]：左侧有一颗子树，其根为 *c*[4]，sum[5]=*c*[4]+*c*[5]。
- sum[9]：左侧有一棵子树，其根为 *c*[8]，sum[9]=*c*[8]+*c*[9]。

**2）点更新**

点更新指修改一个元素的值，例如对 *a*[5]加上一个数 *y*，则需要更新该元素的所有祖先节点，即 *c*[5]、*c*[6]、*c*[8]，令这些节点都加上 *y* 即可，对其他节点都不需要修改。




为什么只修改其祖先节点呢？因为当前节点只和祖先有关系，和其他节点没有关系。

- $c[5]$：存储 $a[5]$的值，修改 $a[5]$加上 $y$，因此 $c[5]$也要加上 $y$。
- $c[6]$：存储 $c[5]$、$a[6]$的和值（$a[5]$、$a[6]$），$a[5]$加上 $y$，$c[6]$也要加上 $y$。
- $c[8]$：存储 $c[4]$、$c[6]$、$c[7]$、$a[8]$的和值（$a[1]$～$a[8]$），$a[5]$加上 $y$，$c[8]$也要加上 $y$。

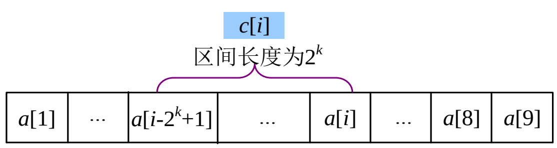
那么这个管理数组（树状数组）是怎么得来的呢？下面详细讲解。

## 2．树状数组的实现

树状数组，又叫作二进制索引树（Binary Indexed Trees），通过二进制分解划分区间。那么 $c[i]$存储的是哪些值？

**1）区间长度**

若 $i$ 的二进制表示末尾有 $k$ 个连续的 0，则 $c[i]$存储的区间长度为 $2^k$，从 $a[i]$向前数 $2^k$ 个元素，即 $c[i]=a[i-2^k+1]+a[i-2^k+2]+\cdots+a[i]$。




例如：$i$=6，6 的二进制表示为 110，末尾有 1 个 0，即 $c[6]$存储的值区间长度为 2（$2^1$），存储的是 $a[5]$、$a[6]$的和值，即 $c[6]=a[5]+a[6]$。

$i$=5，5 的二进制表示为 101，末尾有 0 个 0，即 $c[5]$存储的值区间长度为 1（$2^0$），它存储的是 $a[5]$的值，即 $c[5]=a[5]$。动手试一试，其他值是不是也这样？

怎么得到这个区间的长度呢？若 $i$ 的二进制表示末尾有 $k$ 个连续的 0，则 $c[i]$存储的值区间长度为 $2^k$，换句话说，区间长度就是 $i$ 的二进制表示下最低位的 1 及它后面的 0 构成的数值。例如 $i$=20，其二进制表示为 10100，末尾有两个 0，区间长度为 $2^2$（4），其实就是 10100 最低位的 1 及其后面的 0 构成的数值 100（该数为二进制，其十进制为 4）。

最低位的1

$i$　1 0 1 0 0

怎么得到 100 呢？可以先把 10100 取反，得到 01011，然后加 1 得到 01100，此时，最低位的 1 仍然为 1，而该位前面的其他位与原值相反，因此与原值 10100 进行与运算即可。

最低位的1

$$
\begin{array}{rr}
i & 1\,0\,1\,0\,0 \\
\sim i & 0\,1\,0\,1\,1 \\
 & +\,1 \\ \hline
 & 0\,1\,1\,0\,0 \\
 & \&\,1\,0\,1\,0\,0 \\ \hline
 & 0\,0\,1\,0\,0
\end{array}
$$

- 取反运算（～）：1 变成 0，0 变成 1。
- 与运算（&）：两位都是 1，则为 1，否则为 0。

在计算机中二进制数采用的是补码表示，$-i$ 的补码正好是 $i$ 取反加 1，因此$(-i)\&i$ 就是区间的长度。若将 $c[i]$存储的值区间长度用 lowbit($i$)表示，则 lowbit($i$)=$(-i)\&i$。

**算法代码：**

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

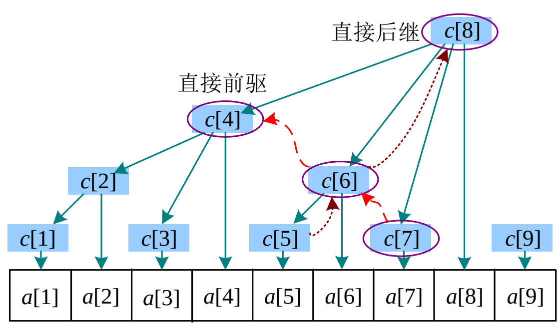
**2）前驱和后继**

直接前驱：$c[i]$的直接前驱为 $c[i-\text{lowbit}(i)]$，即 $c[i]$左侧紧邻的子树的根。

直接后继：$c[i]$的直接后继为 $c[i+\text{lowbit}(i)]$，即 $c[i]$的父节点。

前驱：$c[i]$的直接前驱、其直接前驱的直接前驱等，即 $c[i]$左侧所有子树的根。

后继：$c[i]$的直接后继，其直接后继的直接后继等，即 $c[i]$的所有祖先。




$c[7]$的直接前驱为 $c[6]$，$c[6]$的直接前驱为 $c[4]$，$c[4]$没有直接前驱；$c[7]$的前驱为 $c[6]$、$c[4]$。

$c$[5]的直接后继为 $c$[6]，$c$[6]的直接后继为 $c$[8]，$c$[8]没有直接后继；$c$[5]的后继为 $c$[6]、$c$[8]。

**3）查询前缀和**

前 $i$ 个元素的前缀和 sum[$i$]等于 $c$[$i$]加上 $c$[$i$]的前驱，sum[7]等于 $c$[7]加上 $c$[7]的前驱，$c$[7]的前驱为 $c$[6]、$c$[4]，因此 sum[7]=$c$[7]+$c$[6]+$c$[4]。

**算法代码：**

```
int sum(int i){//求前缀和 a[1]..a[i]
    int s=0;
    for(;i>0;i-=lowbit(i))//直接前驱 i-=lowbit(i);
        s+=c[i];
    return s;
}
```

**4）点更新**

若对 $a$[$i$]进行修改，令 $a$[$i$]加上一个数 $z$，则只需更新 $c$[$i$]及其后继（祖先），即令这些节点都加上 $z$ 即可，不需要修改其他节点。修改 $a$[5]，另其加上 2，则只需 $c$[5]+2，对 $c$[5]的后继分别加上 2，即 $c$[6]+2、$c$[8]+2。

**算法代码：**

```
void add(int i,int z) {//a[i]加上 z
    for(;i<=n;i+=lowbit(i))//直接后继，即父节点 i+=lowbit(i)
        c[i]+=z;
}
```

注意：树状数组的下标从 1 开始，不可以从 0 开始，因为 lowbit(0)=0 时会出现死循环。

**5）查询区间和**

若求区间和值 $a$[$i$]+$a$[$i$+1]+⋯+$a$[$j$]，则求解前 $j$ 个元素的和值减去前 $i$–1 个元素的和值即可，即 sum[$j$]–sum[$i$–1]。

**算法代码：**

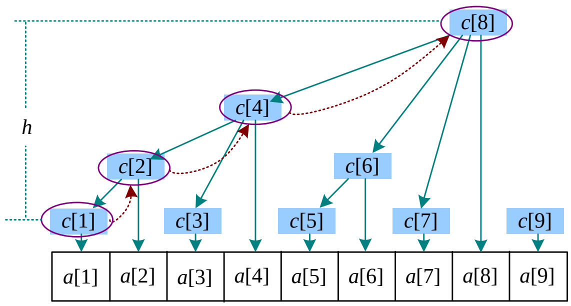
```
int sum(int i,int j) {//求区间和 a[i]..a[j]
    return sum(j)-sum(i-1);
}
```

### 3．算法分析

树状数组是通过二进制分解划分区间的。树状数组的性能与 $n$ 的二进制位数有关，$n$ 的二进制位数为$\lfloor \log n \rfloor$+1，$\lfloor x \rfloor$表示向下取整，即取小于或等于 $x$ 的最大整数。$\lfloor \log 5 \rfloor$=2，5 的二进制位数为 3 位；$\lfloor \log 8 \rfloor$=3，8 的二进制位数为 4。

如何求解树状数组的高度呢？树状数组底层的叶子是 $c$[1]，因此从开始一直找其后继（祖

先）直到树根，就是树状数组的高度。$c[1]$–$c[2^1]$–$c[2^2]$–$c[2^3]$–⋯–$c[n]$，每次都是 2 倍增长，假设 $n=2^x$，则 $x=\log n$，因此树高 $h=O(\log n)$。更新时，从叶子更新到树根，执行的次数不超过树的高度，因此更新的时间复杂度为 $O(\log n)$。




查询前缀和时，需要不停地查找前驱，那么前驱最多有多少个呢？$n$ 的二进制数有 $k=\lfloor\log n\rfloor+1$ 位，在最多的情况下，每一位都是 1，则 $n=$“111⋯1”可以被表示为 $n=2^{k-1}+2^{k-2}+\cdots+2^1+2^0$。7=“111”$=2^2+2^1+2^0$，$c[7]$的前驱为 $c[7-2^0]$、$c[7-2^0-2^1]$、$c[7-2^0-2^1-2^2]$，最后一个为 $c[0]$，表示不存在，因此 $c[7]$的前驱为 $c[6]$、$c[4]$。前驱的个数与 $n$ 的二进制数的位数有关，不超过 $O(\log n)$，因此查询前缀和的时间复杂度为 $O(\log n)$，即树状数组修改和查询的时间复杂度均为 $O(\log n)$。

## 📖 原理 2　多维树状数组

我们已经知道一维树状数组修改和查询的时间复杂度均为 $O(\log n)$，可以扩展为 $m$ 维树状数组，其时间复杂度为 $O(\log^m n)$，对该算法只需加上一层循环即可。二维数组 $a[n][n]$、树状数组 $c[][]$的查询和修改方法如下。

（1）查询前缀和。二维数组的前缀和实际上是从数组左上角到当前位置$(x, y)$矩阵的区间和，在一维数组查询前缀和的代码中加上一层循环即可。

**算法代码：**

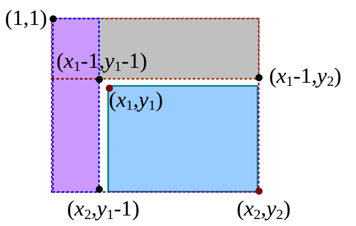
```
int sum(int x,int y) {//求左上角(1,1)到右下角(x,y)矩阵区间和
     int s=0;
     for(int i=x;i>0;i-=lowbit(i))
          for(int j=y;j>0;j-=lowbit(j))
               s+=c[i][j];
     return s;
}
```

（2）更新。若对 $a[x][y]$进行修改（加上 $z$），则在一维数组更新的代码中加上一层循环即可。

**算法代码：**

```
void add(int x,int y,int z) {//a[x][y]加上z
    for(int i=x;i<=n;i+=lowbit(i))
        for(int j=y;j<=n;j+=lowbit(j))
            c[i][j]+=z;
}
```

（3）查询区间和值。对二维数组查询区间和，实际上是求从左上角$(x_1,y_1)$到右下角$(x_2,y_2)$子矩阵的区间和。先求出左上角(1,1)到右下角$(x_2, y_2)$的区间和 $\text{sum}(x_2, y_2)$，然后减去(1,1)到$(x_1-1, y_2)$的区间和 $\text{sum}(x_1-1,y_2)$，再减去(1,1)到$(x_2, y_1-1)$的区间和 $\text{sum}(x_2, y_1-1)$，因为这两个矩阵的交叉区域多减了一次，所以再加回来，加上(1,1)到$(x_1-1, y_1-1)$的区间和 $\text{sum}(x_1-1, y_1-1)$。




**算法代码：**

```
int sum(int x1,int y1,int x2,int y2) {//求左上角(x1,y1)到右下角(x2,y2)子矩阵的区间和
    return sum(x2,y2)-sum(x1-1,y2)-sum(x2,y1-1)+sum(x1-1,y1-1);
}
```

### 4. 树状数组的局限性

树状数组主要用于查询前缀和、区间和及点更新，对点查询、区间修改效率较低。

**前缀和查询：** 求 $a[1]..a[i]$的前缀和，普通数组需要 $O(n)$时间，树状数组需要 $O(\log n)$时间。

**区间和查询：** 求 $a[i]..a[j]$的区间和，普通数组需要 $O(n)$时间，树状数组需要 $O(\log n)$时间。

**点更新：** 修改 $a[i]$加上 $z$，普通数组需要 $O(1)$时间，树状数组需要 $O(\log n)$时间。

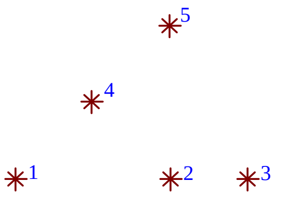
**点查询：** 查找第 $i$ 个元素，普通数组需要 $O(1)$时间，树状数组需要 $O(\log n)$时间（求 $\text{sum}[i]-\text{sum}[i-1]$）。

**区间修改：** 若对一个区间 $a[i]..a[j]$的所有元素都加上 $z$，则普通数组需要 $O(n)$时间，树状数组不能有效操作，只能一个一个地修改和更新，需要 $O(n\log n)$时间。

**减法规则：** 当问题满足减法规则时，例如求区间和 $a[i]..a[j]$，则 $\text{sum}(i,j)=\text{sum}[j]-\text{sum}[i-1]$。当问题不满足减法规则时，例如求区间 $a[i]..a[j]$的最大值，则不可以用 $a[1]..a[j]$的最大值减去 $a[1]..a[i-1]$的最大值，此时可以用线段树解决。

## 训练 1　数星星

**题目描述（POJ2352）：**星星由平面上的点表示，星星的等级为纵横坐标均不超过自己的星星数量（不包括自己）。下图中，5 号星的等级为 3（纵横坐标均不超过 5 号星的星星有 3 颗：1、2 和 4 号）。2 和 4 号星的级别是 1。在该地图上有一颗 0 级星、两颗 1 级星、一颗 2 级星和一颗 3 级星。计算给定地图上每个级别的星星数量。




**输入：**第 1 行包含星星的数量 $N$（$1 \leqslant N \leqslant 15000$）。以下 $N$ 行描述星星的坐标，每行都包含两个整数 $X$、$Y$（$0 \leqslant X$，$Y \leqslant 32000$）。平面上的一个点只可以有一颗星星。以 $Y$ 坐标升序输入，在 $Y$ 坐标相等时以 $X$ 坐标升序输入。

**输出：**输出包含 $N$ 行，第 1 行包含 0 级的星星数量，第 2 行包含 1 级的星星数量……最后一行包含 $N$–1 级的星星数量。

输入样例

```
5
1 1
5 1
7 1
3 3
5 5
```

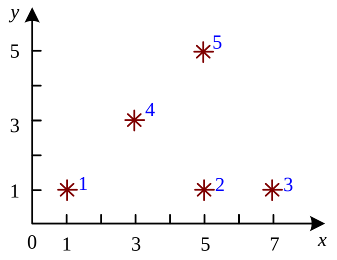
输出样例

```
1
2
1
1
0
```

**提示：**数据量巨大，这里使用 scanf 而不是 cin 来读取数据，避免超出时间限制。

**题解：**每颗星星的等级都为它左下方的星星个数。输入所有星星（按照 $y$ 升序，若 $y$ 相等，则 $x$ 升序）的坐标，依次输出等级 0～$n$–1 的星星数量。

输入样例的地图如下图所示，图中星星旁边的数字为输入顺序，1 号星的左下没有星星，等级为 0；2 号星的左边有 1 颗星星，等级为 1；3 号星的左边有 2 颗星星，等级为 2；4 号星的左下有 1 颗星星，等级为 1；5 号星的左边有 3 颗星星，等级为 3。因此等级为 0 的有 1 个，等级为 1 的有 2 个，等级为 2 的有 1 个，等级为 3 的有 1 个，等级为 4 的有 0 个。




本题看似二维数据，实际上输入数据已经按照 $y$ 升序，也就是说，读到一个点时，当前点的 $y$ 坐标肯定大于或等于已经输入的 $y$ 坐标。如果 $y$ 坐标相等，则 $x$ 坐标肯定大于已经输入的 $x$ 坐标，所以每次只要计算 $x$ 坐标比当前点小的点就行了。该问题的本质是统计 $x$ 坐标前面星星的数量，是前缀和问题。因为数据量较大，暴力穷举会超时，所以可以借助树状数组解决。

注意：给的点坐标从 0 开始，树状数组下标从 1 开始（0 的位置不可用），所以需要在输入 $x$ 坐标时加 1 处理。

### 1．算法设计

（1）依次输入每一个坐标 $x$、$y$，执行 $x$++。

（2）计算 $x$ 的前缀和 sum($x$)，将其作为该星星的等级，用 ans[]数组累计该等级的数量。

（3）将树状数组中 $x$ 的数量加 1。

### 2．算法实现

```
for(int i=0;i<n;i++){
   scanf("%d%d",&x,&y);
   x++;
   ans[sum(x)]++;
   add(x,1);//将 x 的数量 c[x]加 1
}

void add(int i,int val) {//将第 i 个元素增加 val，其后继也要增加
    while(i<=maxn){ //是 x 点的范围，注意不是星星的个数 n
        c[i]+=val;
        i+=lowbit(i);//i 的后继（父节点）
    }
}

int sum(int i) {//前缀和
    int s=0;
    while(i>0){
        s+=c[i];
        i-=lowbit(i);//i 的前驱
```

```
    }
    return s;
}
```

## 训练 2　公路交叉数

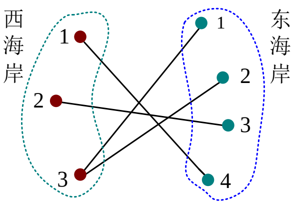
**题目描述（POJ3067）**：东海岸有 $N$ 个城市，西海岸有 $M$ 个城市（$N \leqslant 1000$，$M \leqslant 1000$），将建成 $K$ 条高速公路。每个海岸的城市从北到南编号为 1,2,……每条高速公路都是直线，连接东海岸的城市和西海岸的城市。建设资金由高速公路之间的交叉数决定。两个高速公路最多在一个地方交叉。请计算高速公路之间的交叉数量。

**输入**：输入文件以 $T$ 为开头，表示测试用例的数量。每个测试用例都以 3 个数字 $N$、$M$、$K$ 为开头。下面 $K$ 行中的每一行都包含两个数字，表示由高速公路连接的城市号。第 1 个是东海岸的城市号，第 2 个是西海岸的城市号。

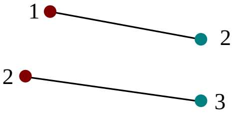
**输出**：对每个测试用例，都单行输出“Test case $x$: $s$”，$x$ 表示输入样例编号，$s$ 表示交叉数。

| 输入样例 | 输出样例 |
| --- | --- |
| 1<br>3 4 4<br>1 4<br>2 3<br>3 2<br>3 1 | Test case 1: 5 |

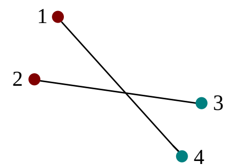
**题解**：根据输入样例分析，一共有 5 个交叉点。




那么，怎么求交叉点呢？首先搞清楚交叉点是怎么产生的。当两条边的城市号都以升序（或降序）形式出现时，不产生交叉点。例如 1 2 和 2 3 不会产生交叉点。




1 4 和 2 3 会产生交叉点，因为西海岸城市 1、2 是升序的，东海岸城市 4、3 是降序的。




因此交叉点的产生原因和逆序对有关系，所以转变为求解逆序对问题。

### 1. 算法设计

（1）对输入的边按照 $x$ 升序排列，若 $x$ 相等，则按 $y$ 升序排列。

（2）检查每条边 $i$，统计 $y$ 的前缀和 sum(e[$i$].$y$)，该前缀和是前面比 $y$ 小的正序数，边数减去正序数，即可得到逆序数 $i$–sum(e[$i$].$y$)，ans 累加逆序数。

（3）将树状数组中 e[$i$].$y$ 的值加 1。

### 2. 完美图解

根据输入样例，其交叉点求解过程如下。

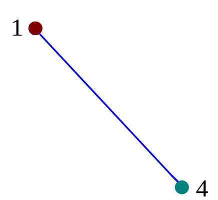
（1）对输入的边按照 $x$ 升序，若 $x$ 相等，则按 $y$ 升序。

排序结果：

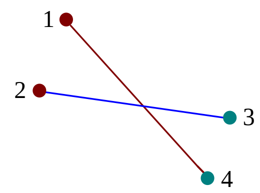
```
1 4
2 3
3 1
3 2
```

（2）按照排序结果检查每条边 $i$，统计 $y$ 的前缀和 sum(e[$i$].$y$)，将 ans 累加 $i$–sum(e[$i$].$y$)。

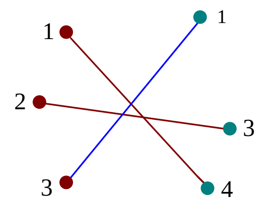
- $i$=0：1 4。sum(4)=0，$i$–sum(4)=0；1 的前缀和为 0，说明 1 前面没有数，因为前面还没有输入边，所以逆序边数量 ans=0。



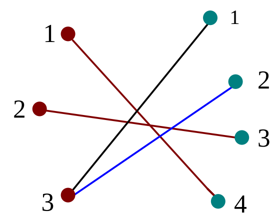

- $i$=1：2 3。sum(3)=0，$i$–sum(3)=1。3 的前缀和为 0，说明 3 前面没有数，所以前面的 1 条边是逆序的，当前边和逆序边会产生交叉点，累加逆序边数量 ans=1。




- $i$=2：3 1。sum(1)=0，$i$–sum(1)=2。1 的前缀和为 0，说明 1 前面没有数，因此前面的两条边是逆序的，当前边和每条逆序边会产生交叉点，累加逆序边数量 ans=3。




- $i$=3：3 2。sum(2)=1，$i$–sum(2)=2；前面的 3 条边已经有 1 条边是正序的，将该边减去，其余两条边是逆序的，当前边和每个逆序边都会产生交叉点，累加逆序边数量 ans=5。



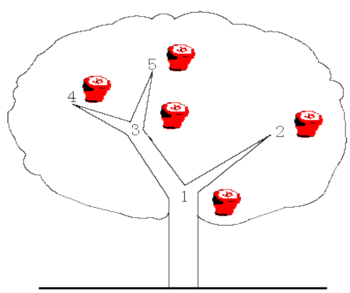

### 3．算法实现

```
void add(int i) {//加 1 操作，参数省略
   while(i<=m){
      ++c[i];
      i+=lowbit(i);
   }
}

int sum(int i){//求前缀和
   int s=0;
   while(i>0){
      s+=c[i];
      i-=lowbit(i);
   }
   return s;
```

```
}

for(int i=0;i<k;i++){
     ans+=i-sum(e[i].y);
     add(e[i].y);
}
```

## 训练 3　子树查询

**题目描述（POJ3321）：**在卡卡的房子外面有一棵苹果树，树上有 $N$ 个叉（编号为 1～$N$，根为 1），它们通过分支连接。苹果在叉上生长，两个苹果不会在同一个叉上生长。一个新的苹果可能会在一个空叉上长出来，卡卡还可能会从树上摘一个苹果作为他的甜点。卡卡想了解一棵子树上有多少苹果。




**输入：**第 1 行包含一个整数 $N$（$N$≤100,000），表示树中叉的数量。以下 $N$–1 行，每行都包含两个整数 $u$ 和 $v$，表示叉 $u$ 和叉 $v$ 通过分支连接。下一行包含整数 $M$（$M$≤100,000）。以下 $M$ 行，每行都包含一个消息，C $x$ 表示改变 $x$ 叉上的苹果状态。若叉上有苹果，则卡卡会选择摘掉它，否则一个新的苹果在这个空叉上长大；Q $x$ 表示查询 $x$ 叉上方子树中的苹果数量，包括 $x$ 叉上的苹果（若存在）。注意：开始时树上长满了苹果。

**输出：**对每个查询，都单行输出答案。

| 输入样例 | 输出样例 |
|---|---|
| 3<br>1 2<br>1 3<br>3<br>Q 1<br>C 2<br>Q 1 | 3<br>2 |

**题解：**本题包含两种操作，一种是点更新，一种是查询以当前节点为根的子树的苹果数量。点更新很简单，那么如何得到以当前节点为根的子树的苹果数量呢？

若将一棵树深度遍历，则记录遍历时当前节点进来和出去时的序号，两个序号之间的节点就是当前节点的子树节点。可以利用 DFS 序将子树转换为序列，然后求解区间和。

## 1. 算法设计

（1）根据输入的分支构建树。

（2）采用深度遍历求树的 DFS 序列，记录进出 $i$ 节点的序号 L[$i$]和 R[$i$]。

（3）Q $x$：查询以 $x$ 节点为根的子树中的苹果数量，只需计算进出 $x$ 节点的区间和[L[$x$],R[$x$]]，即 sum(R[$x$])–sum(L[$x$]–1)。

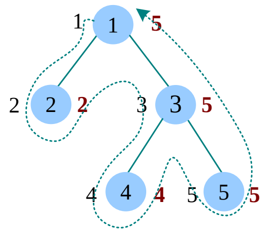
（4）C $x$：若判断 $x$ 节点的值为 1，则在树状数组中点更新–1，否则+1。然后 $a[x]$^=1，进行异或运算，1 变为 0，0 变为 1。

## 2. 完美图解

输入数据如下。

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

（1）构建一棵树，深度优先遍历的 dfs 序列如下图所示。




节点 $i$ 进来和出去时的序号如下：

```
i   L[]  R[]
1   1    5
2   2    2
3   3    5
4   4    4
5   5    5
```

（3）查询或更新操作。

- Q 1：查询以 1 号节点为根的子树中的苹果数量。1 号节点的进出序号为 L[1]=1，R[1]=5，查询[1,5]的区间和，sum(R[1])–sum(L[1]–1)=5–0=5，所以 1 号节点的子树中的苹果数

量为 5。

- Q 3：查询以 3 号节点为根的子树中的苹果数量。3 号节点的进出序号为 L[3]=3，R[3]=5，查询[3,5]的区间和，sum(R[3])–sum(L[3]–1)=5–2=3。所以 3 号节点的子树中的苹果数量为 3。
- C 2：改变 2 号节点的苹果状态。2 号节点有苹果（值为 1），在树状数组中点更新–1，然后 $a$[2]^=1，进行异或运算，1 变为 0，0 变为 1，此时 $a$[2]=0。
- Q 1：查询以 1 号节点为根的子树中的苹果数量，1 号节点的进出序号为 L[1]=1、R[1]=5，查询[1,5]的区间和，sum(R[1])–sum(L[1]–1)=4–0=5。所以 1 号节点的子树中的苹果数量为 4。

### 3．算法实现

```
void dfs(int u,int fa){//DFS 序列
   L[u]=dfn++;
   for(int i=head[u];i;i=E[i].next){
      int v=E[i].v;
      if(v==fa) continue;
      dfs(v,u);
   }
   R[u]=dfn-1;
}
//主函数中的更新和查询操作
if(op[0]=='C'){//更新操作
   if(a[L[v]])
      add(L[v],-1);
   else
      add(L[v],1);
   a[L[v]]^=1;
}
else{//查询操作
   int s1=sum(R[v]);
   int s2=sum(L[v]-1);
   printf("%d\n",s1-s2);
}
```

## 训练 4　矩形区域查询

**题目描述（POJ1195）：**移动电话的基站区域分为多个正方形单元，形成 $S$×$S$ 矩阵，行和列的编号为 0～$S$–1，每个单元都包含一个基站。一个单元内活动手机的数量可能发生变化，因为手机从一个单元移动到另一个单元，或手机开机、关机。编写程序，改变某个单元的活动手机

数量，并查询给定矩形区域中当前活动手机的总数量。

**输入：** 输入和输出均为整数。每个输入都占一行，包含一个指令和多个参数。所有值始终在以下数据范围内。若 $A$ 为负，则可以假设它不会将值减小到零以下。

- 表大小：$1\times1\leqslant S\times S\leqslant1024\times1024$。
- 单元值：$0\leqslant V\leqslant32767$。
- 更新量：$-32768\leqslant A\leqslant32767$。
- 输入中的指令数：$3\leqslant U\leqslant60002$。
- 整个表中的最大电话数：$M=2^{30}$。

| 指　　令 | 参　　数 | 含　　义 |
| --- | --- | --- |
| 0 | $S$ | 初始化 $S\times S$ 矩阵为 0。该指令只会在第一个指令中出现一次 |
| 1 | $X\ Y\ A$ | $(X, Y)$ 单元的活动手机数增加 $A$。$A$ 为正数或负数 |
| 2 | $L\ B\ R\ T$ | 查询$(X, Y)$ 单元的活动手机总数。$L\leqslant X\leqslant R$，$B\leqslant Y\leqslant T$ |
| 3 | | 结束程序。该指令只会在最后一个指令中出现一次 |

**输出：** 对指令 2，单行输出矩形区域中当前活动手机的总数量。

输入样例

```
0 4
1 1 2 3
2 0 0 2 2
1 1 1 2
1 1 2 -1
2 1 1 2 3
3
```

输出样例

```
3
4
```

**题解：** 本题包括单点更新与矩形区间和查询，是非常简单的二维树状数组问题。

### 1. 算法设计

直接采用二维树状数组进行点更新和矩阵区间和查询即可。注意：本题坐标从 0 开始，树状数组下标必须从 1 开始，所以对输入下标做加 1 处理。

### 2. 算法实现

```
void add(int x,int y,int z) {//点更新
   for(int i=x;i<=n;i+=lowbit(i))
      for(int j=y;j<=n;j+=lowbit(j))
         c[i][j]+=z;
}

int sum(int x,int y) {//区间和：左上角(1,1)到右下角(x,y)的矩阵区间和
```

```
    int s=0;
    for(int i=x;i>0;i-=lowbit(i))
        for(int j=y;j>0;j-=lowbit(j))
            s+=c[i][j];
    return s;
}

int sum(int x1,int y1,int x2,int y2) {//求左上角(x1,y1)到右下角(x2,y2)的子矩阵区间和
      return sum(x2,y2)-sum(x1-1,y2)-sum(x2,y1-1)+sum(x1-1,y1-1);
}
```