树状数组（Binary Indexed Tree）

Updated 1548 GMT+8 Mar 20 2025

2024 fall, Complied by Hongfei Yan

树状数组或二叉索引树（英语：Binary Indexed Tree），又以其发明者命名为Fenwick树，最早由Peter M. Fenwick于1994年以A New Data Structure for Cumulative Frequency Tables为题发表。其初衷是解决数据压缩里的累积频率（Cumulative Frequency）的计算问题，现多用于高效计算数列的前缀和， 区间和。

一般来说，如果在查询的过程中元素可能发生改变（例如插入、修改或删除），就称这种查询为在线查询;如果在查询过程中元素不发生改变，就称为**离线查询**。

> 二叉索引树（树状数组）用于处理对固定大小的数组进行以下多种操作的这类问题。
>
> - 前缀操作（求和、求积、异或、按位或等）。注意，区间操作也可以通过前缀来解决。例如，从索引L到R的区间和等于到R（包含R）的前缀和减去到L - 1的前缀和。
> - 更新数组中的一个元素
>
> 这两种操作的时间复杂度均为$O(logN)$。注意，我们需要$O(NlogN)$的预处理时间和$O(N)$的辅助空间。
>
> 让我们考虑以下问题来理解二叉索引树（Binary Indexed Tree, BIT）：
> 我们有一个数组 $arr[0 . . . n - 1]$。我们希望实现两个操作：
>
> 1. 计算前i个元素的和。
> 2. 修改数组中指定位置的值，即设置 $arr[i] = x$，其中 $0 \le i \le n - 1$。
>
> 一个简单的解决方案是从 0 到 i-1 遍历并计算这些元素的总和。要更新一个值，只需执行 $arr[i] = x$。第一个操作的时间复杂度为$O(N)$，而第二个操作的时间复杂度为$O(1)$。另一种简单的解决方案是创建一个额外的数组，并在这个新数组的第i个位置存储前i个元素的总和。这样，给定范围的和可以在$O(1)$时间内计算出来，但是更新操作现在需要$O(N)$时间。当查询操作非常多而更新操作非常少时，这种方法表现良好。
>
> **我们能否在$O(logN)$时间内同时完成查询和更新操作呢?**
> 一种高效的解决方案是使用段树（Segment Tree），它能够在$O(logN)$时间内完成这两个操作。
> 另一种解决方案是二叉索引树（Binary Indexed Tree，也称作Fenwick Tree），同样能够以$O(logN)$的时间复杂度完成查询和更新操作。与段树相比，二叉索引树所需的空间更少，且实现起来更加简单。

lowbit 运算

二进制中一个经典应用是 lowbit 运算，即 `lowbit(x) = x & (-x)`。

整数的二进制表示常用的方式之一是使用补码

补码是一种表示有符号整数的方法，它将负数的二进制表示转换为正数的二进制表示。补码的优势在于可以使用相同的算术运算规则来处理正数和负数，而不需要特殊的操作。

在补码表示中，最高位用于表示符号位，0表示正数，1表示负数。其他位表示数值部分。

具体将一个整数转换为补码的步骤如下：

1. 如果整数是正数，则补码等于二进制表示本身。
2. 如果整数是负数，则需要先将其绝对值转换为二进制，然后取反，最后加1。等价于把二进制最右边的1的左边的每一位都取反。

例如，假设要将 -12 转换为补码：

1. 12的二进制表示为00001100。
2. 将其取反得到11110011。
3. 加1得到11110100，这就是 -12 的补码表示。

通过`lowbit(x) = x & (-x)`就是取 x 的二进制最右边的1和它右边所有的0，因此它一定是2的幂次，即1、2、4、8等。

对 x = 12 = $(00001100)_2$，有 -x = $(11110100)_2$ ，x & (-x) = 4

对 x= 6 = $(110)_2$，有 -x = $(010)_2$，x & (-x) = 2

表示方式

树状数组（Binary Indexed Tree，BIT）用数组形式表示。它其实仍然是一个数组，并且与 sum 数组类似，是一个用来记录和的数组，只不过它存放的不是前 i 个整数之和，而是在 i 号位之前（含i号位）lowbit(i) 个整数之和。树状数组的大小等于输入数组的大小，记为n。在下面的代码中，为了便于实现，使用n+1的大小。

如下图 所示，数组A是原始数组，有 A[1]~ A[16]共 16个元素；数组 C是树状数组，其中 C[i]存放数组 A 中i号位之前 lowbit(i) 个元素之和。显然，C[i]的覆盖长度是 lowbit(i)（也可以理解成管辖范围），它是2的幂次，即1、2、4、8等。

需要注意的是，树状数组仍旧是一个平坦的数组，画成树形是为了让存储的元素更容易观察。

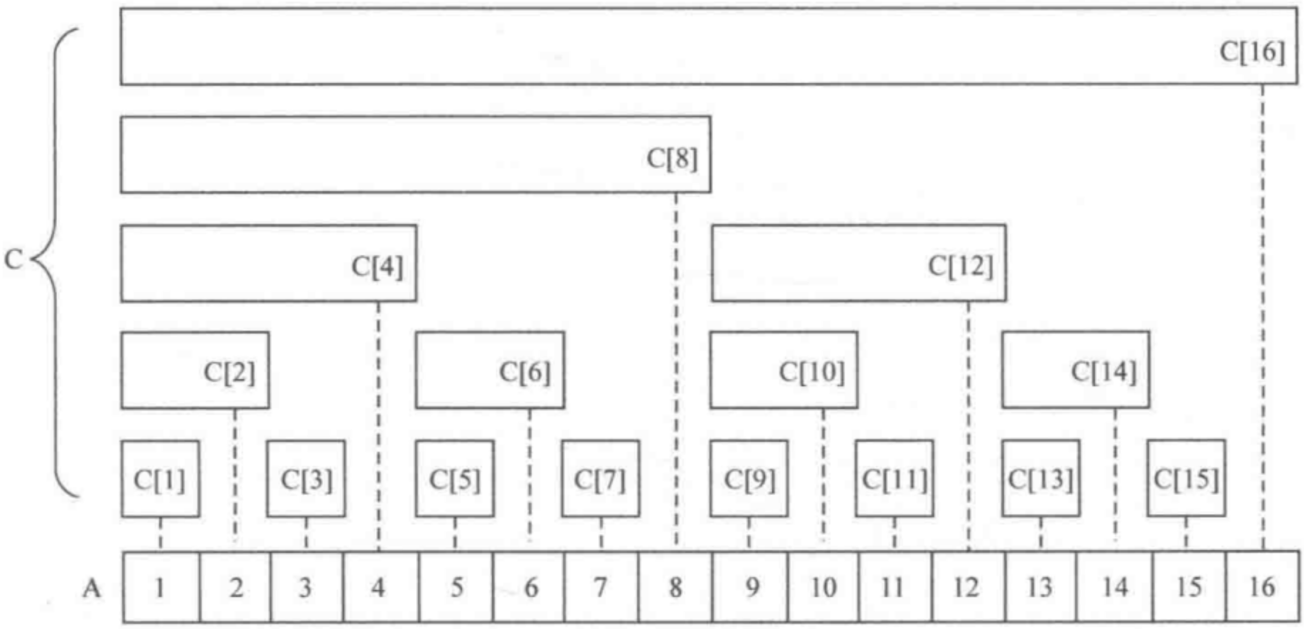


图 树状数组定义图

```
C[1] = A[1]                            (长度为 lowbit(1) = 1)
C[2] = A[1] + A[2]                     (长度为 lowbit(2) = 2)
C[3] = A[3]                            (长度为 lowbit(3) = 1)
C[4] = A[1] + A[2] + A[3] + A[4]       (长度为 lowbit(4) = 4)
C[5] = A[5]                            (长度为 lowbit(5) = 1)
C[6] = A[5] + A[6]                     (长度为 lowbit(6) = 2)
C[7] = A[7]                            (长度为 lowbit(7) = 1)
C[8] = A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7] + A[8]  (长度为 lowbit(8) = 8)
```

树状数组的定义非常重要，特别是“C[i]的覆盖长度是 lowbit(i)”这点；另外，树状数组的下标必须从1开始。接下来思考一下，在这样的定义下，
怎样解决下面两个问题：

① 设计函数 get_sum(x)，返回前x个数之和 A[1]+...+ A[x]。

② 设计函数 update_bit(x,v)，实现将第x个数加上一个数v的功能，即 A[x]+= v。

先来看第一个问题，即如何设计函数 get_sum(x)，返回前x个数之和。不妨先看个例子。假设想要查询 A[1]+...+A[14]，那么从树状数组的定义出发，它实际是什么东西呢? 回到上图，很容易发现 A[1]+...+A[14] = C[8]+C[12]+ C[14]。又比如要查询 A[1]+...A[11]，从图中同样可以得到 A[1]+...+A[11] = C[8]+C[10]+ C[11]。那么怎样知道 A[1]+...+ A[x]对应的是树状数组中的哪些项呢？事实上这很简单。记 SUM(1,x) = A[1]+......+A[x]，由于 C[x]的覆盖长度是 lowbit(x)，因此

C[x] = A[x-lowbit(x)+1]+...+ A[x]

于是可以得到

```
SUM(1,x) = A[1] +···+ A[x]
        =A[1] +···+ A[x-lowbit(x)] + A[x-lowbit(x)+1] +···+ A[x]
        =SUM(1,x-lowbit(x)) + C[x]
```

这样就把 SUM(1,x)转换为 SUM(1,x-lowbit(x))了。

接着就能写出 get_sum 函数了，其中BITTree是树状数组。

```python
def bit_sum(BIT, i):
    s = 0
    i += 1  # index in BIT[] is 1 more than the index in arr[]

    while i > 0:  # Traverse ancestors of BIT[index]
        s += BIT[i]
        i -= i & (-i)  # Move index to parent node
    return s
```

由于 lowbit(i)的作用是定位i的二进制中最右边的1，因此 `i = i- lowbit(i)` 事实上是不断把i的二进制中最右边的1置为0的过程。所以 get_sum 函数的 for 循环执行次数为x的二进制中1的个数。一个数n的二进制表示中设置位的数量是O(logn)。也就是说，get_sum 函数的时间复杂度为 $O(logN)$。从另一个角度理解,结合图会发现，get_sum 函数的过程实际上是在沿着一条不断左上的路径行进（可以想一想 get_sum(14)跟 get_sum(11)的过程）。于是由于“树”高是 $O(logN)$级别,因此可以同样得到 get_sum 函数的时间复杂度就是 $O(logN)$。另外，如果要求数组下标在区间[x,y]内的数之和，即 A[x] + A[x+1] +...+ A[y]，可以转换成 get_sum(y) - get_sum(x-1)来解决，这是一个很重要的技巧。

接着来看第二个问题，即如何设计函数 update(x,v)，实现将第x个数加上一个数v的功能。
来看两个例子。假如要让 A[6]加上一个数 v，那么就要寻找树状数组C中能覆盖了 A[6]的元素，让它们都加上 v。也就是说，如果要让 A[6]加上 v，实际上是要让C[6]、C[8]、C[16]都加上 v。同样，如果要将 A[9]加上一个数 v,实际上就是要让 C[9]、C[10]、C[12]、C[16]都加上 v。于是问题又来了——想要给 A[x]加上v时，怎样去寻找树状数组中的对应项呢?

要让 A[x]加上 v，就是要寻找树状数组 C 中能覆盖 A[x]的那些元素，让它们都加上 v。而从图 1中直观地看，只需要总是寻找离当前的“矩形”C[x]最近的“矩形”C[y]，使得 C[y]能够覆盖 C[x]即可。例如要让 A[6]加上 v，就从 C[6]开始找起：离 C[6]最近的能覆盖 C[6]的“矩形”是 C[8]，离 C[8]最近的能覆盖 C[8]的“矩形”是 C[16]，于是只要把 C[6]、C[8]、C[16]都加上v即可。

那么，如何找到距离当前的 C[x]最近的能覆盖 C[x]的 C[y]呢？首先，可以得到一个显然的结论：lowbit(y)必须大于 lowbit(x)（不然怎么覆盖呢......）。于是问题等价于求一个尽可能小的整数 a，使得 lowbit(x+a)>lowbit(x)。显然，由于 lowbit(x)是取x的二进制最右边的1的位置，因此如果 lowbit(a) < lowbit(x)，lowbit(x+ a)就会小于 lowbit(x)。为此 lowbit(a)必须不小于 lowbit(x)。接着发现，当a取 lowbit(x)时，由于x和a的二进制最右边的1的位置相同,因此x+a会在这个1的位置上产生进位，使得进位过程中的所有连续的1变成0，直到把它们左边第一个0置为1时结束。于是lowbit(x+a)>lowbit(x)显然成立,最小的a就是 lowbit(x)。于是 update 函数的做法就很明确了，只要让x不断加上 lowbit(x)，并让每步的 C[x]都加上 v，直到x超过给定的数据范围为止。代码如下：

```python
def bit_update(BIT, n, i, v):
    i += 1  # index in BITree[] is 1 more than the index in arr[]

    while i <= n:  # Traverse all ancestors and add 'val'
        BIT[i] += v
        i += i & (-i)  # Update index to that of parent
```

更新函数需要确保所有包含arr[i]在其范围内的BIT节点都被更新。我们通过不断向当前索引添加其最后一位设置位对应的十进制数，在BIT中循环遍历这些节点。

实现

首先将BIT[]中的所有值初始化为0。然后对所有的索引调用bit_update()函数。

```python
# Binary Indexed Tree

def bit_sum(BIT, i):
    """计算树状数组 BIT 从索引 1 到 i 的前缀和"""
    s = 0
    while i > 0:
        s += BIT[i]
        i -= i & (-i)  # 回溯至祖先节点
    return s


def bit_update(BIT, i, v):
    """在树状数组 BIT 中更新索引 i 处的值 v"""
    while i < len(BIT):
        BIT[i] += v
        i += i & (-i)  # 回溯至祖先节点


# Constructs and returns a Binary Indexed Tree for given array of size n.
def construct(arr, n):
    BIT = [0] * (n + 1)
    for i in range(n):  # Store the actual values in BIT[] using bit_update()
        bit_update(BIT, i + 1, arr[i])

    return BIT


arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
BIT = construct(arr, len(arr))
print(f'BIT: ', *BIT)
print("Sum of elements in arr[0..5] is " + str(bit_sum(BIT, 5)))
arr[3] += 6
bit_update(BIT, 3, 6)
print(f'BIT: ', *BIT)
print("Sum of elements in arr[0..5]" +
      " after update is " + str(bit_sum(BIT, 5)))

```

Output

```
BIT:  0 1 3 3 10 5 11 7 36 9 19 11 42 13 27 15 136
Sum of elements in arr[0..5] is 15
BIT:  0 1 3 9 16 5 11 7 42 9 19 11 42 13 27 15 142
Sum of elements in arr[0..5] after update is 21
```

Time Complexity: $O(NlogN)$
Auxiliary Space: $O(N)$

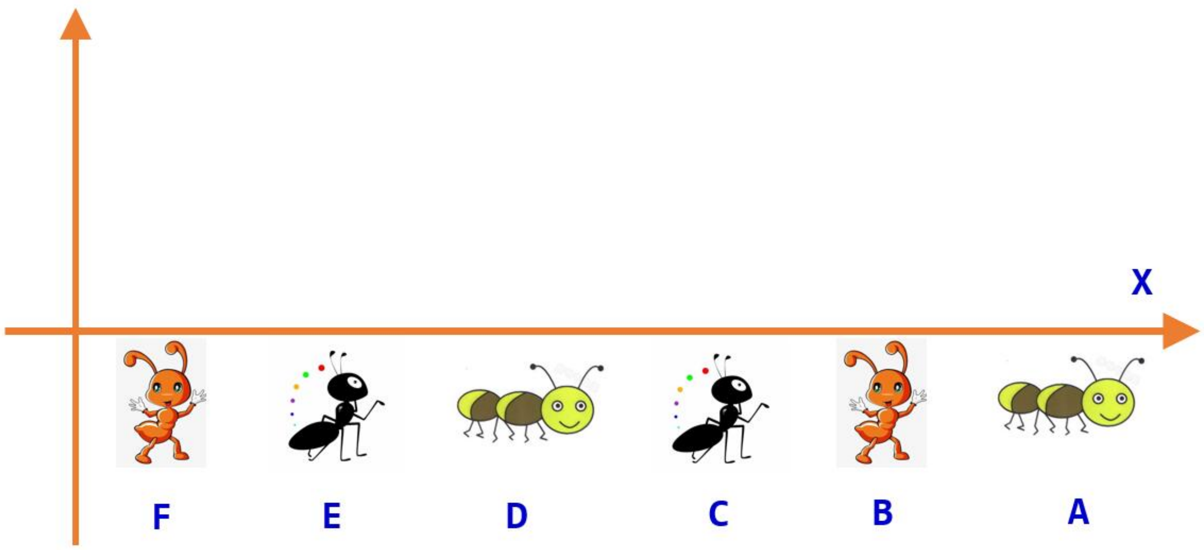
Can we extend the Binary Indexed Tree to computing the sum of a range in O(Logn) time?
Yes. rangeSum(l, r) = get_sum(r) – get_sum(l-1).

References:
http://en.wikipedia.org/wiki/Fenwick_tree

示例20018:蚂蚁王国的越野跑

BIT, http://cs101.openjudge.cn/practice/20018/

为了促进蚂蚁家族身体健康，提高蚁族健身意识，蚂蚁王国举行了越野跑。假设越野跑共有N个蚂蚁参加，在一条笔直的道路上进行。N个蚂蚁在起点处站成一列，相邻两个蚂蚁之间保持一定的间距。比赛开始后，N个蚂蚁同时沿着道路向相同的方向跑去。换句话说，这N个蚂蚁可以看作x轴上的N个点，在比赛开始后，它们同时向X轴正方向移动。假设越野跑的距离足够远，这N个蚂蚁的速度有的不相同有的相同且保持匀速运动，那么会有多少对参赛者之间发生“赶超”的事件呢？此题结果比较大，需要定义long long类型。请看备注。



输入

第一行1个整数N。
第2... N +1行：N 个非负整数，按从前到后的顺序给出每个蚂蚁的跑步速度。对于50%的数据，2<=N<=1000。对于100%的数据，2<=N<=100000。

输出

一个整数，表示有多少对参赛者之间发生赶超事件。

样例输入

```
5
1
5
10
7
6

5
1
5
5
7
6
```

样例输出

```
7

8
```

提示

我们把这5个蚂蚁依次编号为A,B,C,D,E，假设速度分别为1,5,5,7,6。在跑步过程中：B,C,D,E均会超过A，因为他们的速度都比A快；D,E都会超过B,C，因为他们的速度都比B,C快；D,E之间不会发生赶超，因为速度快的起跑时就在前边；B,C之间不会发生赶超，因为速度一样，在前面的就一直在前面。

考虑归并排序的思想。

此题结果比较大，需要定义long long类型，其输出格式为printf("%lld",x);
long long，有符号 64位整数，所占8个字节(Byte)
-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

```python
# 张清州 24化学学院
def bit_sum(BIT, i):
    """计算树状数组 BIT 从索引 1 到 i 的前缀和"""
    s = 0
    while i > 0:
        s += BIT[i]
        i -= i & (-i)  # 回溯至祖先节点
    return s


def bit_update(BIT, i, v):
    """在树状数组 BIT 中更新索引 i 处的值 v"""
    while i < len(BIT):
        BIT[i] += v
        i += i & (-i)  # 回溯至祖先节点
```

```python


# 读取输入并进行离散化
n = int(input())
values = [int(input()) for _ in range(n)]

# 离散化：建立值到索引的映射
sorted_vals = sorted(set(values))
value_to_index = {v: i + 1 for i, v in enumerate(sorted_vals)}

# 初始化树状数组
BIT = [0] * (len(sorted_vals) + 1)
count = 0

# 计算逆序对
for v in values:
    index = value_to_index[v]
    count += bit_sum(BIT, index - 1)  # 查询比当前值小的元素个数
    bit_update(BIT, index, 1)  # 在树状数组中记录当前值出现次数

print(count)
```