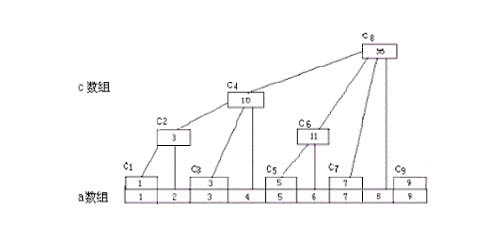
**[树状数组](http://idearush.ycool.com/post.1753778.html)**

树状数组是一个查询和修改复杂度都为log(n)的数据结构，假设数组a[1...n]，那么查询a[1] + …… + a[i]　的时间是log级别的，而且是一个在线的数据结构，支持随时修改某个元素的值，复杂度也为log级别。  
来观察一下这个图：  
  
令这棵树的结点编号为C1，C2……Cn。令每个结点的值为这棵树的值的总和，那么容易发现：  
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  
……  
C16 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8 + A9 + A10 + A11 + A12 + A13 + A14 + A15 + A16  
……  
C2^n=a1+a2+….+a2^n  
  
对于序列a，我们设一个数组C定义C[t] = a[t – 2^k + 1] + … + a[t]，k为t在二进制下末尾0的个数。   
K的计算可以这样:   
2^k=t and (t xor (t-1))   
以6为例   
               (6)10=(0110)2   
xor    6-1=(5)10=(0101)2   
                        (0011)2   
and          (6)10=(0110)2   
                        (0010)2   
  
  
这里有一个有趣的性质:  
设节点编号为x，那么这个节点管辖的区间为2^k（其中k为x二进制末尾0的个数）个元素。因为这个区间最后一个元素必然为Ax，所以很明显：  
Cn = A(n – 2^k + 1) + …… + An  
算这个2^k有一个快捷的办法，定义一个函数如下即可：  
int lowbit(int x){  
return x & (x ^ (x – 1)); //return x & (-x);  
}  
  
当想要查询一个SUM(n)时，可以依据如下算法即可：  
step1:　令sum = 0，转第二步；  
step2:　假如n <= 0，算法结束，返回sum值，否则sum = sum + Cn，转第三步；  
step3: 令n = n – lowbit(n)，转第二步。  
  
  
可以看出，这个算法就是将这一个个区间的和全部加起来，为什么是效率是log(n)的呢？以下给出证明：  
n = n – lowbit(n)这一步实际上等价于将n的二进制的最后一个1减去。而n的二进制里最多有log(n)个1，所以查询效率是log(n)的。  
  
那么修改呢，修改一个节点，必须修改其所有祖先，最坏情况下为修改第一个元素，最多有log(n)的祖先。所以修改算法如下（给某个结点i加上x）：  
step1: 当i > n时，算法结束，否则转第二步；  
step2: Ci = Ci + x， i = i + lowbit(i)转第一步。  
  
i = i +lowbit(i)这个过程实际上也只是一个把末尾1补为0的过程。  
//修改过程必须满足减法规则！

树状数组是一个可以很高效的进行区间统计的数据结构。在思想上类似于线段树，比线段树节省空间，编程复杂度比线段树低，但适用范围比线段树小。

以简单的求和为例。设原数组为a[1..N]，树状数组为c[1..N]，其中c[k] = a[k-(2^t)+1] + ... + a[k]。比如c[6] = c[5] + c[6]。也就是说，把k表示成二进制1\*\*\*10000，那么c[k]就是1\*\*\*00001 + 1\*\*\*00010 + ... + 1\*\*\*10000这一段数的和。设一个函数lowestbit(k)为取得k的最低非零位，容易发现，根据上面的表示方法，从a[1]到a[k]的所有数的总和即为sum[k] = c[k] + c[k-lowestbit(k)] + c[k-lowestbit(k)-lowestbit(k-lowestbit(k))] + ... 于是可以在logk的时间内求出sum[k]。当数组中某元素发生变化时，需要改动的c值是c[k],c[k+lowestbit(k)], c[k+lowestbit(k)+lowestbit(k+lowestbit(k))] ... 这个复杂度是logN (N为最大范围)

扩展到多维情况：以二维为例，用c[k1][k2]表示c[k1-(2^t1)+1][k2-(2^t2)+1] + ... + c[k1][k2]的总和。可以用类似的方法进行处理。复杂度为(logn)^k (k为维数)

树状数组相比线段树的优势：空间复杂度略低，编程复杂度低，容易扩展到多维情况。劣势：适用范围小，对可以进行的运算也有限制，比如每次要查询的是一个区间的最小值，似乎就没有很好的解决办法。

多维情况的几道题目:

POJ 2155 Matrix  
URAL 1470 UFOs

其中POJ 2155是一道很不错的题目，表面上看，这题的要求似乎和树状数组的使用方法恰好相反，改变的是一个区间，查询的反而是一个点。实际上可以通过一个转化巧妙的解决。

首先对于每个数A定义集合up(A)表示{A, A+lowestbit(A), A+lowestbit(A)+lowestbit(A+lowestbit(A))...} 定义集合down(A)表示{A, A-lowestbit(A), A-lowestbit(A)-lowestbit(A-lowestbit(A)) ... , 0}。可以发现对于任何A<B，up(A)和down(B)的交集有且仅有一个数。

于是对于这道题目来说，翻转一个区间[A,B]（为了便于讨论先把原问题降为一维的情况），我们可以把down(B)的所有元素的翻转次数+1，再把down(A-1)的所有元素的翻转次数-1。而每次查询一个元素C时，只需要统计up(C)的所有元素的翻转次数之和，即为C实际被翻转的次数。

实际实现时，由于只考虑奇偶，因此无须统计确切的翻转次数。另外，如果翻转up(A)和up(B+1)，查询down(C)，也是同样的效果。这种方法可以很容易地扩展到二维情况。比起线段树、四分树之类的常规思路，无论编程复杂度还是常数速度上都有很大优势。

PS:  
int lowbit(int t)  
{   
    return t & (-t);  
}   
void ...()  
{    ...  
    pos+=lowbit(pos); //如果pos=0，那么这个地方pos将永远是0  
}