# 树状数组

树状数组在区间求和问题上有大用，其三种复杂度都比线段树要低很多……有关区间求和的问题主要有以下三个模型（以下设A[1..N]为一个长为N的序列，初始值为全0）：

**（1）“改点求段”型，即对于序列A有以下操作：**

【1】修改操作：将A[x]的值加上c；

【2】求和操作：求此时A[l..r]的和。

这是最容易的模型，不需要任何辅助数组。树状数组中从x开始不断减lowbit(x)（即x&(-x)）可以得到整个[1..x]的和，而从x开始不断加lowbit(x)则可以得到x的所有前趋。代码：

void ADD(int x, int c)  
{  
     for (int i=x; i<=n; i+=i&(-i)) a[i] += c;  
}  
int SUM(int x)  
{  
    int s = 0;  
    for (int i=x; i>0; i-=i&(-i)) s += a[i];  
    return s;  
}

操作【1】：ADD(x, c);

操作【2】：SUM(r)-SUM(l-1)。  
  
**（2）“改段求点”型，即对于序列A有以下操作：**

【1】修改操作：将A[l..r]之间的全部元素值加上c；

【2】求和操作：求此时A[x]的值。  
  
这个模型中需要设置一个辅助数组B：B[i]表示A[1..i]到目前为止共被整体加了多少（或者可以说成，到目前为止的所有ADD(i, c)操作中c的总和）。

则可以发现，对于之前的所有ADD(x, c)操作，当且仅当x>=i时，该操作会对A[i]的值造成影响（将A[i]加上c），又由于初始A[i]=0，所以有A[i] = B[i..N]之和！而ADD(i, c)（将A[1..i]整体加上c），将B[i]加上c即可——只要对B数组进行操作就行了。

【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。

   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】

这样就把该模型转化成了“改点求点”型，只是有一点不同的是，SUM(x)不是求B[1..x]的和而是求B[x..N]的和，此时只需把ADD和SUM中的增减次序对调即可（模型1中是ADD加SUM减，这里是ADD减SUM加）。代码：

void ADD(int x, int c)  
{  
     for (int i=x; i>0; i-=i&(-i)) b[i] += c;  
}  
int SUM(int x)  
{  
    int s = 0;  
    for (int i=x; i<=n; i+=i&(-i)) s += b[i];  
    return s;  
}

操作【1】：ADD(l-1, -c); ADD(r, c);  
  
操作【2】：SUM(x)。

**（3）“改段求段”型，即对于序列A有以下操作：**  
【1】修改操作：将A[l..r]之间的全部元素值加上c；  
  
【2】求和操作：求此时A[l..r]的和。  
  
这是最复杂的模型，需要两个辅助数组：B[i]表示A[1..i]到目前为止共被整体加了多少（和模型2中的一样），C[i]表示A[1..i]到目前为止共被整体加了多少的总和（或者说，C[i]=B[i]\*i）。  
  
对于ADD(x, c)，只要将B[x]加上c，同时C[x]加上c\*x即可（根据C[x]和B[x]间的关系可得）；  
  
而ADD(x, c)操作是这样影响A[1..i]的和的：若x<i，则会将A[1..i]的和加上x\*c，否则（x>=i）会将A[1..i]的和加上i\*c。也就是，A[1..i]之和 = B[i..N]之和 \* i + C[1..i-1]之和。  
这样对于B和C两个数组而言就变成了“改点求段”（不过B是求后缀和而C是求前缀和）。  
另外，该模型中需要特别注意越界问题，即x=0时不能执行SUM\_B操作和ADD\_C操作！代码：

void ADD\_B(int x, int c)  
{  
     for (int i=x; i>0; i-=i&(-i)) B[i] += c;  
}  
void ADD\_C(int x, int c)  
{  
     for (int i=x; i<=n; i+=i&(-i)) C[i] += x \* c;  
}  
int SUM\_B(int x)  
{  
    int s = 0;  
    for (int i=x; i<=n; i+=i&(-i)) s += B[i];  
    return s;  
}  
int SUM\_C(int x)  
{  
    int s = 0;  
    for (int i=x; i>0; i-=i&(-i)) s += C[i];  
    return s;  
}  
inline int SUM(int x)  
{  
    if (x) return SUM\_B(x) \* x + SUM\_C(x - 1); else return 0;  
}

操作【1】：  
ADD\_B(r, c); ADD\_C(r, c);  
if (l > 1) {ADD\_B(l - 1, -c); ADD\_C(l - 1, -c);}  
操作【2】：SUM(r) - SUM(l - 1)。