树状数组：一个学到自闭的数据结构。

这里、今天、现在暂时记录下树状数组的讲解，希望可以在记录的过程中加深对这一结构的理解，并且寄希望于在日后的做题或者阅读中能有更深的理解。

算法真的是一个考验智商的东西，我好像没有。

笔记参考自博客https://www.cnblogs.com/xenny/p/9739600.html

什么是树状数组？

就是用数组来模拟树形结构。

解决的问题？

用来解决大部分基于区间上的更新和求和问题。

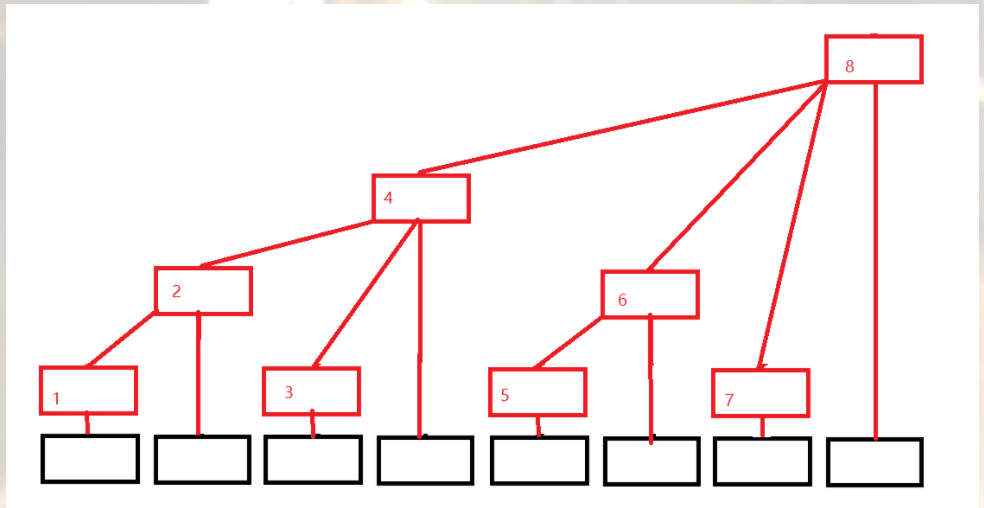
和线段树的区别？

树状数组可以解决的问题用线段树都可以解决，即线段树的功能包含树状数组，但是树状数组拥有比线段树更小的系数。

树状数组的优点和缺点？

优点是更新和查询的复杂度都是O(lgn)，而且比线段树系数少很多，且比传统数组快。

树状数组介绍



注意相比于二叉树的区别，二叉树是只有两个子节点，而树状数组为了记录更多的信息，用于区间操作，子节点数量符合以下规律：假设A[i]代表原来数组，C[i]代表树状数组，则每个位置存的是子节点的和，即

C[1] = A[1]

C[2] = A[1] + A[2]

C[3] = A[3]

C[4] = A[1] + A[2] + A[3] + A[4]

可以发现如下规律：

C[i] = A[i-2^k+1] + A[i-2^k+2] +…+A[i] //其中k为i的二级制中从最低位到最高位连续0的长度，专业名词称为lowbit，用来计算2^k。即C[i]中是lowbit(i)位相邻的数字之和。

lowbit(x)求出x的二进制表达式中最低位的1所对应的值。

比如6的二进制110，lowbit(6)=2.

常见算法有两种：

1. x&(x^(x-1))。这个算法很好理解，x-1将最低位的1变为0，并将右边的0都变为1，异或得到最低位1到最低位的0都为1，并且高位都是0的二进制。再与原数字求与操作，得到仅包含最低位1的二进制。得到结果。
2. x&(-x)。这个算法有些复杂，利用负数的存储特性，以原码的补码+1形式存储，需要分如下情况讨论：
   1. x为0时，0&0=0
   2. x为奇数时，最低位为1，取反后最低位为0，加1后没有进位，则除最低位之外求与结果为0，而最低位为1，所以结果为1，即最低位。
   3. x为偶数且为2的m次方时，x二进制表示只有最高位为1，右边都是0，则x取反+1后，依旧是最高位为1，右边都是0的形式，x&(-x)得到的就是x。
   4. x为偶数且不为2的m次方时，最低位有1且位置小于m，取反加1后，最低位的1依旧为1，但高位均取反，求与操作之后仅剩原最低位仍为1，求解结果为最低位1对应的十进制结果。

具有如上lowbit知识之后，就可以建立树状数组，此处以求和为例，即求区间[start, end]的元素和。

首先已知C[i] = A[i-2^k+1]+…+A[i]，那么如果更新A[i]的值，则包含有A[i]的C[i], C[i+2^k1],C[i+2^k1+2^k2]…均需要同理更新，因为i+2^k每次均需要从1加到i+2^k，一定包含A[i]，且k1,k2表示的是当前i的lowbit值。而如果要求区间的和，例如[l, r]的和，则已知C[r]表示[1,r]的和，C[l]表示[1,l]的和，则C[r]-C[l-1]即为所求。

def lowbit(x):

return x&(-x)

# 在位置i上加k，即单点更新，可用于建树阶段

def update(i, k):

while i <= n:

c[i] += k

i += lowbit(i)

# 求[1,i]的和

def get\_sum(i):

ans = 0

while i > 0:

ans += c[i]

i -= lowbit(i)

return ans

上述是最常见的单点更新，区间查询问题。模板例题：有n个兵营，每个兵营会有人事变动，需要知道连续一段兵营的总人数。

此外还有如下三种情况：

1. 单点更新、单点查询。使用普通一维数组即可。
2. 区间更新，单点查询。

这个问题是，如果把区间[l, r]内的所有值全部加上k，然后查询单个点的值。如果采用上述建树方法，必须[l, r]内所有的值都更新一次，时间复杂度为O(nlgn)，远超我们预期。所以，在这里就需要使用差分建树。

假设A[0]=0, D[i]=A[i]-A[i-1]，则有A[i] = ，即前面i项的差值和，例如对于数组

A[] = [1, 2, 3, 5, 6, 9]

D[] = [1, 1, 1, 2, 1, 3]，

如果将区间[2,5]内值加2，则变成了

A[] = [1, 4, 5, 7, 8, 9]

D[] = [1, 3, 1, 2, 1, 1]，

即，某一个区间值[l, r]改变时，区间内的差值和区间外不相邻的差值是不变的，并且只有D[l]和D[r+1]改变。所以在进行区间修改时，只需要更新对应l和r+1位置处的值，即可维护和上述同样的树状数组。利用这个性质对数组D建立树状数组。

def lowbit(x):

return x&(-x)

def update(i, k):

while i <= n:

c[i] += k

i += lowbit(i)

def get\_sum(i):

ans = 0

while i > 0:

ans += c[i]

i -= lowbit(i)

return ans

需要注意这里在建树的时候，需要首先根据输入数组求出差分数组D，然后对数组D进行建树。即在update时传入D[i]。如果要对区间[l,r]加k，需要首先调用update(l, k)和update(r+1, -k)才能完成区间的更新。

这样的实现方式把更新整个区间的值变为更新两个点。

1. 还有一种情况时，区间更新、区间查询。

利用差分思想，

则可以得到A[1]+A[2]+…+A[n]=(D[1]) + (D[1]+D[2]) +…+(D[1]+…D[n])=n\*D[1] + (n-1)\*D[2]+…+D[n]=n\*(D[1]+…+D[n])-(0\*D[1]+1\*D[i]+…+n\*D[n])。

所以上式变为

那么现在就可以维护两个数组，一个是sum1[i]是建立在D[i]上的树状数组，一个是sum2[i]是建立在D[i]\*(i-1)的树状数组。

def lowbit(x):

return x&(-x)

def update(i, k):

# 因为每次更新sum2时的i-1是应该不变的，而经过lowbit之后i会发生改变

x = i

while i <= n:

sum1[i] += k

sum2[i] += k \* (x-1)

i += lowbit(i)

def get\_sum(i):

ans = 0

x = i

# 也需要记录i，因为总结果是n乘连续和。

while i > 0:

ans = ans + x \* sum1[i] - sum2[i]

i -= lowbit(i)

return ans

注意这里和第二种情况一样，在更新区间[l,r]时，要update(l,k)和update(r+1, -k)。

除了求区间和之外，求区间最值问题也可以用树状数组实现。

拓展题：使用树状数组求解逆序对。

结合树状数组的单点更新，区间查询功能，按照如下思路求解逆序对：一个数组的逆序对为所有数字前大于该数字的个数之和。即遍历到一个数字时，将该数字应该在的位置加1，表示这里在有序时应该为该数字，使用树状数组记录小于该数字的数字个数，（可以依靠索引实现），然后已经插入的数字减去小于该数字的个数就是大于该数字的个数，即为逆序对的个数。

时间复杂度为（nlgn），每个点需要遍历一次，且进行一次更新。

例如[5,3,4,2,1]，使用一个数组a表示数字应该出现位置的记录。

1. 5, 3, 4, 2, 1

0, 0, 0, 0, 1

表示5应该在第5位的位置，所以a[5]+=1。然后求区间[1,5]的个数和为1，且插入了1个数，所以逆序对为1-1=0

1. 5, 3, 4, 2, 1

0, 0, 1, 0, 1

表示3在a[3]，a[3]+=1。求[1,3]个数和为1，但已经插入了2个数，说明插入的数字是在3后面，所以逆序对个数为2-1=1

3）5, 3, 4, 2, 1

0, 0, 1, 1, 1

表示a[4]+=1.求[1,4]和为2，插入3个数，逆序对个数为3-2=1.

其余同理不再介绍。

代码为：

def lowbit(x):

return x&(-x)

def update(i, k):

while i <= n:

c[i] += k

i += lowbit(i)

def get\_sum(i):

ans = 0

while i > 0:

ans += c[i]

i -= lowbit(i)

return ans

def reverse\_pair():

ans = 0

for i in range(1, n+1):

update(nums[i]+1, 1)

ans += i - get\_sum(nums[i]+1)

return ans

上述为一维树状数组，tree[x]记录的是[1,x]区间内的特征值如最值或者和，即右端点为x，长度为lowbit(x)的区间信息。而在二维树状数组中，可以类似的定义tree[x][y]记录的是右下角为(x,y)，高为lowbit(x)，宽为lowbit(y)的区间信息。

1. 单点修改，区间查询。这种情况类似于一维，只是在更新lowbit(x)时，需要将对应的y也进行更新。因为x向右下移动时，变化的除了x还有y。

def lowbit(x):

return x&(-x)

# 将点(x, y)加上k

def update(x, y, k):

origin\_y = y

while x <= n:

y = origin\_y

while y <= n:

c[x][y] += z

y += lowbit(y)

x += lowbit(x)

# 求(1,1)到(x,y)包含的矩形区域的和

def get\_sum(x, y):

ans = 0

origin\_y = y

while x > 0:

y = origin\_y:

while y > 0:

ans += c[x][y]

y -= lowbit(y)

x -= lowbit(x)

return ans

1. 区间修改，单点查询

上述一维数组利用差分性质D[i] = A[i] – A[i-1]，得到对区间修改时差分数组仅会对区间的两个边界进行修改的结论。那么同样，二维数组也应该利用差分性质，只不过二维前缀和为

sum[i][j] = sum[i-1][j]+sum[i][j-1]-sum[i-1][j-1]+a[i][j]。

那么差分数组d[i][j]表示a[i][j]-(a[i-1][j]+a[i][j-1]-a[i-1][j-1])，这样可以得到a[i][j] = 求和d[i][j]（i=1,n;j=1,n）。

例如矩阵

1 4 8

6 7 2

3 9 5

的差分矩阵为

1 3 4

5 -2 -9

-3 5 1

如果此时修改(1,1)到（2，2）的区间内的值加2

则原矩阵为

3 6 8

8 9 2

3 9 5

差分矩阵变为

3 3 2

5 -2 -9

-5 5 3

可以得到，如果修改(x1,y1)到(x2,y2)的区间内的值加x，则差分矩阵c[x1][y1]和c[x2+1][y2+1]值加x，c[x1][y2+1]和c[x2+1][y1]加-x。

def lowbit(x):

return x&(-x)

# 将点(x, y)加上k

def update(x, y, k):

origin\_y = y

while x <= n:

y = origin\_y

while y <= n:

c[x][y] += z

y += lowbit(y)

x += lowbit(x)

# 求(1,1)到(x,y)包含的矩形区域的和

def get\_sum(x, y):

ans = 0

origin\_y = y

while x > 0:

y = origin\_y:

while y > 0:

ans += c[x][y]

y -= lowbit(y)

x -= lowbit(x)

return ans

# 修改区间(x1,y1)到(x2,y2)的值加k

def range\_modify(x1, x2, y1, y2, k):

update(x1, y1, k)

update(x2+1, y2+1, k)

update(x2+1, y1, -k)

update(x1, y2+1, -k)

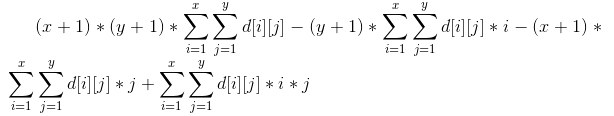
1. 区间修改，区间查询

类比此前一维数组的区间修改和区间查询，点(x,y)的前缀和表示为

，即也是利用差分的思想，只是扩展到二维空间，这样x和y都需要两次遍历求和。同样参考一维树状数组的想法，统计每个坐标值出现的次数，例如求解a[x][y]的前缀和，需要统计d[h][k]出现多少次时，d[1][1]出现x\*y次，d[1][2]出现x\*(y-1)次，d[h][k]出现(x-h+1)\*(y-k+1)次。那么求前缀和的式子就变成了



展开上式就得到



此时得到了和一维数组类似的形式，所以新开4个树状数组来分别维护d[i][j], d[i][j]\*I, d[i][j]\*j和d[i][j]\*i\*j，称为t1, t2, t3, t4

def lowbit(x):

return x&(-x)

# 将点(x, y)加上k

def update(x, y, k):

origin\_y = y

origin\_x = x

while x <= n:

y = origin\_y

while y <= n:

t1[x][y] += k

t2[x][y] += k \* x

t3[x][y] += k \* y

t4[x][y] += k \* x \* y

y += lowbit(y)

x += lowbit(x)

def range\_update(x1, y1, x2, y2, k):

update(x1, y1, k)

update(x1+1, y2, -k)

update(x2, y1+1, -k)

update(x2+1, y2+1, k)

def get\_sum(x, y):

ans = 0

origin\_y = y

origin\_x = x

while x > 0:

y = origin\_y

while y > 0:

ans += (origin\_x + 1) \* (origin\_y + 1) \* t1[x][y] - (origin\_y + 1) \* t2[x][y] - (origin\_x + 1) \* t3[x][y] + t4[x][y]

y -= lowbit(y)

x -= lowbit(x)

return ans

def range\_sum(x1, y1, x2, y2):

return get\_sum(x2, y2) - get\_sum(x2, y1-1) - get\_sum(x1-1, y2) + get\_sum(x1-1, y1-1)