数据结构和算法实习

郭炜

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

树状数组

树状数组的概念


我们是谁？
树状数组！
我们任务是什么？
单点更新，区间求和！
我们有多快？
Log（N）！

树状数组的定义

- 对于数组 a，我们设一个数组C
  - C[i] = a[i – lowbit(i) + 1] + a[i – lowbit(i) + 2] +… + a[i]
  - lowbit(x) : x 的二进制表示形式留下最右边的1，其他位都变成0
  - i 从1开始算！ C[0]和a[0]没用
- C即为a的树状数组。

求lowbit(x)

lowbit(x) : x 的二进制表示形式留下最右边的1，其他位都变成0

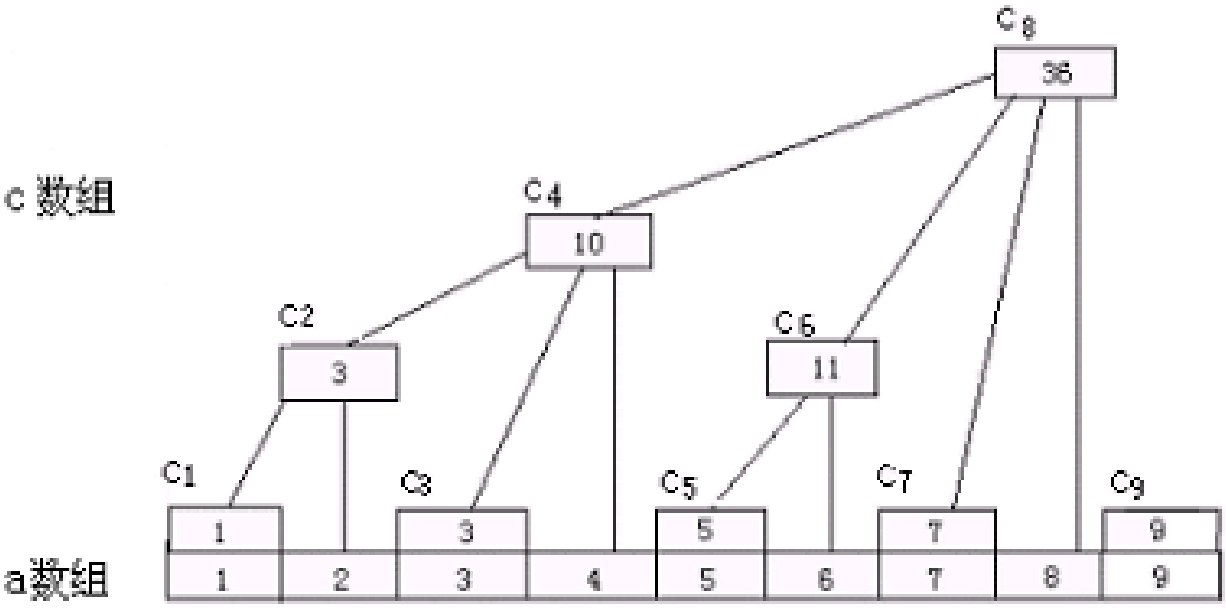
```
lowbit(x)= x & (x^(x-1))   = x &(-x)
```

C包含哪些项看上去没有规律

C[i] = a[i-lowbit(i)+1] + …+ a[i]

- 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

树状数组图示



树状数组的作用

树状数组用于解决单个元素经常修改，
而且还反复求不同的区间和的情况

用O(logN)时间求a数组任意区间的和 a[i] + a[i+1] + … + a[j]

代价：更新一个元素a[i]所花时间也是O(logN)
因为a[i]更新会导致C数组一些元素更新

树状数组性能证明:
求和

为何求区间和复杂度是O(logN)

设 sum(k) = a[1]+a[2]+...+a[k]

则 a[i] + a[i+1] + ... + a[j] = sum(j)-sum(i-1)

有了树状数组C，sum(k)就能在O(logN)时间内求出，故

sum(j)-sum(i-1) 也能在O(logN)时间内求出，即区间和
a[i] + a[i+1] + ... + a[j] 能在O(logN)时间求出

为何求sum(k)可在logN时间完成

根据C的定义，可证明:

$sum(k) = C[n_1]+C[n_2] + \ldots + C[n_m]$

$n_m = k,\ n_{i-1} = n_i - lowbit(n_i)$,

n_1大于0 且 $n_1 - lowbit(n_1)$ 等于0

如：sum(6) = C[4]+C[6]

sum(k) 项数m决定了求区间和的时间复杂度

sum(k)的项数

- $sum(k) = C[n_1]+C[n_2] + \ldots + C[n_m]$　其中 $n_m= k$， $n_{i-1} = n_i$ - lowbit(n_i)
- lowbit(x) : x的二进制表示形式留下最右边的1，其他位都变成0
- n_i - lowbit(n_i)　：就是 n_i的二进制去掉最右边的1
- k 的二进制里最多有 $\log_2 k$ （向上取整）个1
- 故 sum(k)最多$\log_2 k$ （向上取整）项

证明sum(k)的构成

$sum(k) = C[n_1]+C[n_2] + ...+ C[n_m]$

$n_m = k$, $n_{i-1} = n_i$ - lowbit(n_i) ,

n_1大于0 且 n_1 – lowbit(n_1) 等于0

复习：

$C[i] = a[i\text{-}lowbit(i)+1] + ...+ a[i]$

i - lowbit(i)+1： i 把最右边的1去掉，然后再加1

证明sum(k)的构成

$C[n_m] = a[n_m\text{-lowbit}(n_m)+1] + \ldots + a[n_m]$ $\quad (n_m = k)$

$C[n_{m-1}] = a[n_{m-1}\text{-lowbit}(n_{m-1})+1] + \ldots + a[n_{m-1}]$

$= a[n_{m-1}\text{-lowbit}(n_{m-1})+1] + \ldots + a[n_m\text{-lowbit}(n_m)]$

$C[n_{m-2}] = a[n_{m-2}\text{-lowbit}(n_{m-2})+1] + \ldots + a[n_{m-2}]$

$= a[n_{m-1}\text{-lowbit}(n_{m-1})+1] + \ldots + a[n_{m-1}\text{-lowbit}(n_{m-1})]$

……..

$C[n_1] = a[n_1\text{-lowbit}(n_1)+1] + \ldots + a[n_1]$

$= a[1] + \ldots + a[n_1]$

(因 $n_1\text{-lowbit}(n_1)$ 必须等于0，否则就还需要 $C[n_1\text{-lowbit}(n_1)]$了)

树状数组性能证明:
更新

更新单个a元素

更新一个a元素（如a[2]），C也要跟着更新。
复杂度取决于C里有几项包含被更新的a元素

- 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

更新单个a元素

如果a[i]更新，那么有且仅有以下的几项需要更新：

$C[n_1], C[n_2], \ldots C[n_m]$

其中，$n_1 = i$，$n_{p+1} = n_p + lowbit(n_p)$

$n_m + lowbit(n_m)$ 必须大于 a 的元素个数 N, n_m 小于等于N

同理，总的来说更新一个元素的时间，也是O(logN)的

更新单个a元素

如果a[i]更新，那么有且仅有以下的几项需要更新：

$C[n_1], C[n_2], \ldots C[n_m]$

其中，$n_1 = i$，$n_{p+1} = n_p + \text{lowbit}(n_p)$

$n_m + \text{lowbit}(n_m)$ 必须大于 a 的元素个数 N, n_m 小于等于N

同理，总的来说更新一个元素的时间，也是O(logN)的

因为，x+lowbit(x) 把x的最低位的1往左推进了至少1位

更新单个a元素

- **证明以下命题:**

如果a[i]更新了，那么有且仅有以下的几项需要更新：

$C[n_1], C[n_2], \ldots C[n_m]$　　其中，$n_1 = i$，$n_{p+1} = n_p + lowbit(n_p)$

第一步: 证明上述各项需要更新

1) a[i]更新-> C[i]必须更新， 因为 C[i] = a[i-lowbit(i)+1] + …+ a[i]

2) C[k+lowbit(k)] 的起始项不晚于C[k]的起始项, 所以,若C[k]包含 a[i],
则 C[k+lowbit(k)]也包含a[i],即 C[k] 需要更新-> C[k+lowbit(k)]也需要更新

更新单个a元素

证明 C[k+lowbit(k)] 的起始项不晚于C[k]的起始项

- C[k] = a[k-lowbit(k)+1] + ...
- C[k+lowbit(k)] = a[k+lowbit(k) – lowbit(k+lowbit(k))+1] + ...
- k – lowbit(k) >= k+lowbit(k) – lowbit(k+lowbit(k))

  易证, 因为 lowbit(k+lowbit(k)) >= 2 * lowbit(k)

综上所述: C[k+lowbit(k)] 的起始项不晚于C[k]的起始项

更新单个a元素

第二步: 证明 若a[i]更新, 除下面项以外的项，都不需要更新

$C[n_1], C[n_2], \ldots C[n_m]$ 其中，$n_1 = i$，$n_{p+1} = n_p + lowbit(n_p), n_m <= N$

1) 若 k < i, C[k]的最后一项是a[k], 显然 C[k]不需要更新

2) 命题： 对任何k (x < k < x +lowbit(x)), C[k]的起始项都在a[x]后面

- C[k] = a[k-lowbit(k)+1] + … + a[k]
- 只要证明 k-lowbit(k)+1比 x 大即可
- 因 x< k <x+lowbit(x) ,假设 x 的最右边的1是从右到左从0开始数的第n位，那么x+lowbit(x)就是将x的低n位全变成1后，再加1。那么k一定是从第n位到最高位都和x相同，但是低n位比x大(即k低n位中有1，因x低n位全是0)
- k-lowbit(k) +1就是k去掉最右边的1，然后再加1，那当然还是比x大

更新单个a元素

第二步: 证明 若a[i]更新, 除下面项以外的项，都不需要更新
$C[n_1]$, $C[n_2]$, …$C[n_m]$ 其中，$n_1 = i$ ，$n_{p+1} = n_p + lowbit(n_p)$, $n_m <= N$

3) 根据 2) 命题 :

对任何 $n_p < k < n_{p+1}$ $(p>=1)$, C[k]起始项在$a[n_p]$ 右边，因$n_p>= i$, 因此C[k]不需更新

构建树状数组

- 初始状态下由a构建树状数组C的时间复杂度是O(N)

因为：C[k] = sum(k) – sum(k-lowbit(k))， 所有sum(k)可在O(N)求出

- $sum(k) = C[n_1]+C[n_2] + \ldots + C[n_{m-1}] + C[n_m]$ $(n_m = k)$
- $n_{m-1} = k - lowbit(k)$
- $sum(k-lowbit(k)) = C[n_1]+C[n_2] + \ldots + C[n_{m-1}]$

树状数组总结

对原始数组a:

- 树状数组 C[i] = a[i-lowbit(i)+1] + … + a[i]　　连续
- sum(K) = a[1]+a[2]+ …a[K] = C[n_1]+C[n_2] …+ C[K]　　间隔

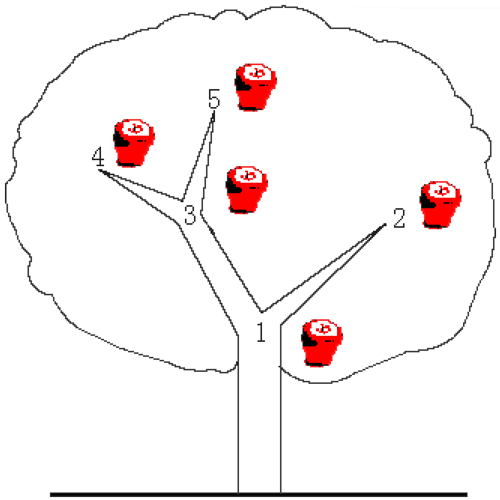
  $n_{i-1} = n_i - \text{lowbit}(n_i)$, 项数最多为 log(K), 各项不相交
- 对任意一个元素a[i]，最多有logN个C元素包含a[i]

  C[n_1], C[n_2] …,　$n_1 = i$, $n_i = n_{i-1} + \text{lowbit}(n_{i-1})$
- 建数组：　O(N)
- 区间求和：O(logN)
- 更新单个原始数组元素：　O(logN)

例题: Apple Tree


新疆安集海大峡谷

例题1: POJ 3321 Apple Tree



一棵苹果树，每个分叉点及末梢可能有苹果（最多1个），每次操作可以摘掉一个苹果，或让一个苹果新长出来。反复进行多次操作，以及随时查询某个分叉点往上的子树里，一共有多少个苹果。（分叉点数： 100,000 ）

例题1： POJ 3321 Apple Tree

根据题意，一开始时，
所有能长苹果的地方都有苹果，1号点是树根

Sample Input		Sample Output
3	//三个节点	3
1 2	//2是1儿子	2
1 3	//3是1儿子	
3	//3个询问	
Q 1	//1号点有几个苹果	
C 2	//改变2号点苹果数量	
Q 1		

例题1： POJ 3321 Apple Tree

- 用邻接表存图（每个节点 V 对应于一个vector，vector里放和V有边相连的点）
- 单点更新，反复求和，似乎可以用树状数组做
- 原始数组和区间在哪里?

例题1： POJ 3321 Apple Tree

- 用邻接表存图（每个节点 V 对应于一个vector，vector里放和V有边相连的点）
- 单点更新，反复求和，似乎可以用树状数组做
- 原始数组和区间在哪里?

◆做一次dfs，记下每个节点的开始时间Start[i]和结束时间End[i]，则i节点的所有子孙的开始时间和结束时间都应位于Start[i]和End[i]之间。

◆每个节点对应于2个时间点。时间点序列A为:1， 2， 3. ……， 则一棵子树(假设根为i) 上的所有节点所对应的时间点正好是A上一个连续的区间Start[i] --- End[i]，且该子树上每个节点对应于该区间中的两个元素。

例题1： POJ 3321 Apple Tree

- 用邻接表存图（每个节点 V 对应于一个vector，vector里放和V有边相连的点）
- 单点更新，反复求和，似乎可以用树状数组做
- 原始数组和区间在哪里?

◆做一次dfs，记下每个节点的开始时间Start[i]和结束时间End[i]，则i节点的所有子孙的开始时间和结束时间都应位于Start[i]和End[i]之间。

◆每个节点对应于2个时间点。时间点序列A为:1，2，3. ……，则一棵子树(假设根为i) 上的所有节点所对应的时间点正好是A上一个连续的区间Start[i] – End[i]，且该子树上每个节点对应于该区间中的两个元素。

- 改变苹果数目就是单点更新，统计一棵子树上面的所有苹果数，就是区间求和。

更通用的树状数组

树状数组更通用的定义

对原始数组a:

- 树状数组 C[i] 对应于 a[i-lowbit(i)+1] , a[i-lowbit(i)+2] … a[i]
  C[i]可用于存放和这些对应的a元素相关的值，比如和，最大值，最小值
- 令F(K) 对应于 a[1], a[2] …a[K]，即对应于 $C[n_1], C[n_2]$ …C[K]
  $n_{i-1} = n_i - \text{lowbit}(n_i)$ ，项数最多为 log(K), 各项不相交
  F(K) 可以代表与区间 a[1], a[2] …a[K] 相关的某值，如和，最大值，最小值
- 对任意一个元素a[i]，最多有logN个C元素包含a[i]
  $C[n_1], C[n_2] \ldots, \quad n_1 = i, n_i = n_{i-1} + \text{lowbit}(n_{i-1})$
- 建数组： O(NlogN)
- 区间求值(如求和，求区间最大最小值）： O(logN)
- 更新单个原始数组元素： O(logN)

树状数组更通用的定义

若 F([s,e]) 可以由 F([1,s-1])和F([1,e])推导出来，才有可能用树状数组求 F([s,e])

例如，树状数组不能做单点更新，区间求最大值。

但如果区间总是从1开始，则可以。

例题2：nlogn求 最长上升子序列 (百练2757:最长上升子序列)

一个数的序列b_i，当$b_1 < b_2 < ... < b_S$的时候，我们称这个序列是上升的。对于给定的一个序列($a_1, a_2, ..., a_N$)，我们可以得到一些上升的子序列($a_{i1}, a_{i2}, ..., a_{iK}$)，这里$1 <= i_1 < i_2 < ... < i_K <= N$。

比如，对于序列(1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如(1, 7), (3, 4, 8)等等。这些子序列中最长的长度是4，比如子序列(1, 3, 5, 8).

你的任务，就是对于给定的序列，求出最长上升子序列的长度。

例题2：nlogn求 最长上升子序列

样例输入

7 1 7 3 5 9 4 8

样例输出

4

例题2：nlogn求 最长上升子序列 (LIS)

- 对原数组 a 从小到大排序得到新数组n。n[i]在a中原来的位置记为 n[i].pos（相同元素，如何排序？）
- 设定树状数组C, C[i]表示a中终点位于 a[i-lowbit(i)+1] 到 a[i] 之间（含两端)的LIS的长度，会不断更新
- 用LIS(i)表示a中，以 a[i]为终点的LIS的长度。开始所有的 LIS(i)都是0
- 从小到大扫描 n 一遍 , 扫描到n[i]时:
  1> LIS(n[i].pos) 的值设为 query(n[i].pos) + 1
  query(k)表示：目前a[1]到a[k]之间（含两端）的LIS的长度。查询最多log(k)项C数组元素即可完成
  2> 更新所有包含a[n[i].pos] 的C元素（最多logN项)
- 最大的C元素，就是答案 (所有上述数组下标从1开始)

例题2：nlogn求 最长上升子序列 (LIS)

- 从小到大扫描 n 一遍 , 扫描到n[i]时:

  1> LIS(n[i].pos) 的值设为 query(n[i].pos) + 1

  解释:

  - query求出的，是目前a数组中从a[1]到 a[n[i].pos] 这一段中的LIS（称为LISA)的长度。
  - 此时a[n[i].pos] 还没被看过，自然不会在LISA里面。
  - 看 n 是按从小到大看的，看到 n[i]的时候，所发现的所有LIS，包括 LISA 都不包含比 n[i] 大的元素。
  - 如果LISA里面的元素，都比n[i] 小，那么加上 n[i]，即 a[n[i].pos]，自然就能形成一个更长的LIS。如果LISA里面有和n[i] 相等的元素，咋办？？？

例题2：nlogn求 最长上升子序列 (LIS)

- 从小到大扫描 n 一遍，扫描到n[i]时:

  1> LIS(n[i].pos) 的值设为 query(n[i].pos) + 1

  解释:

  - query求出的，是a数组中从a[1]到 a[n[i].pos] 这一段中的LIS（称为LISA)的长度。
  - 此时a[n[i].pos] 还没被看过，自然不会在LISA里面。
  - 看 n 是按从小到大看的，看到 n[i]的时候，所发现的所有LIS，包括 LISA 都不包含比 n[i] 大的元素。
  - 如果LISA里面的元素，都比n[i] 小，那么加上 n[i]，即 a[n[i].pos]，自然就能形成一个更长的LIS。如果LISA里面有和n[i] 相等的元素，咋办？？？

  - 要确保 LISA里面不能有和 n[i] 相等的元素
  - 解决办法：n里面的元素相等的情况下，按 pos 从大到小排

二维树状数组

二维树状数组

- 原始数组a和树状数组C都是二维的
- $C[x][y] = \sum \{a[i][j]\}$

  $x\text{-lowbit}[x]+1 <= i <= x$

  $y\text{-lowbit}[y]+1 <= j <= y$

- $Sum[x][y] = \sum \{C\ [i][j]\}$ (从[1,1] 到[x,y]这个a矩阵里的所有元素的和)

  $i_1=x,\ i_2 = i_1\text{-lowbit}(i_1),\ \ldots i_k = i_{k-1}\text{-lowbit}(i_{k-1})\ \ldots..\ \ i_k > 0$

  $j_1=y,\ j_2 = j_1\text{-lowbit}(j_1),\ \ldots j_k = j_{k-1}\text{-lowbit}(j_{k-1})\ \ldots..\ \ j_k > 0$

- 用于快速求数字子矩阵的和,更新和查询的时间复杂度都是log(n)*log(m)， 建C数组复杂度O(m*n) (n,m分别为两维的大小)