2016 年 4 月 15 日

307. Range Sum Query - Mutable

Given an integer array nums, find the sum of the elements between indices i and j (i ≤ j), inclusive.

The update(i, val) function modifies nums by updating the element at index i to val.

Example:

Given nums = [1, 3, 5]

sumRange(0, 2) -> 9

update(1, 2)

sumRange(0, 2) -> 8

Note:

The array is only modifiable by the update function.

You may assume the number of calls to update and sumRange function is distributed evenly.

思路：

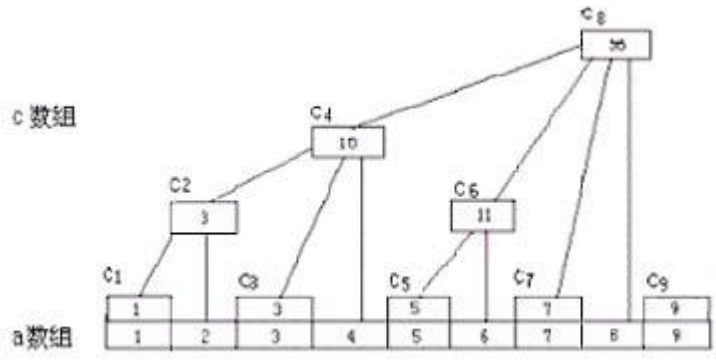
树状数组主要用于查询任意两位之间的所有元素之和，但是每次只能修改一个元素的值；可

以在 log(n)的复杂度下进行范围修改

树状数组：

http://blog.csdn.net/int64ago/article/details/7429868

1.这个图：我们只维护 c 数组，标号一定要从 1 开始



2.c 数组与 A 数组的对应关系：C[n] 负责的 A 的元素个数为 lowbit(n):n&(-n)

C[n]一定监管 A[n]

如 c1:01 末尾 0 个数为 0，因此负责 2^0=1 个 A[1]

C2: 10 。。。。为 1，因此负责 2^1=2 个 A[1]和 A[2]

c3:11 .....0.....2^0=1 个 A[3]

插入时 ： 由子节点向父亲节点依次更新

求和时：由父亲节点向子节点依次往下累加

重要关系式：

1）lowbit 是建立树状数组父子节点的桥梁，c[i+lowbit(i)]是找 c[i]的父节点,c[i-lowbit(i)]是找 c[i]的子节点

2）sum[k] = c[k] + c[k-lowbit[k]] +c[k-lowbit[k-lowbitk]]+....

```cpp
class NumArray {
private:
    vector<int> C;
    vector<int> m_nums;
public:
    NumArray(vector<int> &nums) {
        int length = nums.size();
        C.resize(length+1,0);//resize
        m_nums = nums;//deepcopy
        for(int i=0;i<nums.size();i++)
            add(i+1,nums[i]);
    }

    int lowbit(int n)//树状数组父节点到子节点之间的步长差
    {
        return n&(-n);
    }

    void add(int k,int val)
    {
        while(k<=C.size()-1)
        {
            C[k]+=val;
            k+=lowbit(k);//找父节点
        }
    }

    int getSum(int k)//sum for [0,n]
    {
        int sum = 0;
        while(k>0)
        {
            sum+=C[k];
            k-=lowbit(k);//找子节点
        }
        return sum;
    }

    void update(int i, int val) {
        int diff = val-m_nums[i];
        m_nums[i] = val;
        if(diff)
            add(i+1,diff);
    }

    int sumRange(int i, int j) {
        return getSum(j+1)-getSum(i);
    }
};
```

312.Burst Balloons

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get nums[left] * nums[i] * nums[right] coins. Here left and right are adjacent indices of i. After the burst, the left and right then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:

(1) You may imagine nums[-1] = nums[n] = 1. They are not real therefore you can not burst them.

(2) 0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100

Example:

Given [3, 1, 5, 8]

Return 167

```
    nums = [3,1,5,8] --> [3,5,8] -->   [3,8]   -->  [8]  --> []
  coins =  3*1*5      +  3*5*8    +  1*3*8      + 1*8*1   = 167
```

思路：

动态规划:

opt(i,j)表示在（i,j）范围内(不含第 i,j 个气球)能得到的最大分数。

给定 nums 数组，前后补 1 后(nums[0]=1,nums[nums.size(0-1]=1)，则 nums 数组所能得到的最大分数就是 opt(0,nums.size()-1);

动规方程:

opt(i,j)= max(opt(i,j),opt(i,m)+opt(m,j)+A[i]*A[m]*A[j])（不含 i,j）

含义：opt(i,j)的最大值为从(i,j)中挑一个气球 m 最后爆炸，则 opt(i,m)和 opt(m,j)可分为两个独立的子问题，气球 m 最后爆炸得分为 nums[i]*nums[m]*nums[j]

关键：如果气球 m 不是最后一个爆炸，则 opt(i,m)和 opt(m,j)不独立！

注意:i,j 是当前气球范围的边界(可能是 nums[0],nums[nums.size()-1]，也可能是以其它气球作为边界)

由 opt(i,j)的含义可知：

初始化时,opt(i,i)=0,opt(i,i+1)=0---范围内没有气

球　opt(i,i+2)=nums[i]*nums[i+1]*nums[i+2](只有一个气球)

观察递推表达式：

(i,m) -> (i,j)

|

(m,j)

j 从下往上计算，i 从左往右，m 从左往右

```
class Solution {
public:
    int maxCoins(vector<int>& nums) {
        if(nums.empty())
            return 0;
        nums.insert(nums.begin(),1);
        nums.push_back(1);
        vector<vector<int>>opt(nums.size(),vector<int>(nums.size(),0));
        //initialization
        for(int i=0;i<nums.size();i++)
        {
            for(int j=i;j<nums.size();j++)
            {
                if(j<i+2)//opt[i][i]=0,opt[i][i+1]=0//中间无气球，只有气球爆炸才能得分
                    opt[i][j] = 0;
                else if(j==i+2)//中间只有一个气球
                    opt[i][j] = nums[i]*nums[i+1]*nums[j];
            }
        }
        //
        for(int row=nums.size()-4;row>=0;row--)//从下面的行开始求解
        {
            for(int col = row+3;col<nums.size();col++)//列序：从前往后
            {
                opt[row][col] = INT_MIN;
                for(int m=row+1;m<=col-1;m++)//选择(row,col)范围内的一个气球最后爆炸
                {
                    opt[row][col] = max(opt[row][col],opt[row][m]+opt[m][col]+nums[row]*nums[m]*nums[col]);
                }

            }
        }
        return opt[0][nums.size()-1];
    }
};
```

---

315. Count of Smaller Numbers After Self

You are given an integer array nums and you have to return a new counts array. The counts array has the property where counts[i] is the number of smaller elements to

the right of nums[i].

Example:

Given nums = [5, 2, 6, 1]

To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
Return the array [2, 1, 1, 0].

思路：

1）用二叉排序树进行计数，每个 BST 节点记录以当前节点为根节点的子树上<当前节点的有多少个，=当前节点的有多少个；这样就有点儿类似于以二叉排序树的根节点对其进行分段统计

2）将输入的节点逆序插入 BST，当插入节点的同时对<=当前节点的进行计数

举例：9 5 2 6 4

1)插入 4

4(lt_cnt=0,equal_cnt=1)

---

ret = 0

2)插入 6

4(lt_cnt=0,equal_cnt=1)

\

6(lt_cnt=0,equal_cnt=1)

---

ret = 4(equal_cnt=1)+4(lt_cnt=0) =1

3)插入 2

4(lt_cnt=1,equal_cnt=1)

/ \

2(lt_cnt=0,equal_cnt=1) 6(lt_cnt=0,equal_cnt=1)

---

ret = 0

4)插入 5

4(lt_cnt=1,equal_cnt=1)

/ \

2(lt_cnt=0,equal_cnt=1) 6(lt_cnt=1,equal_cnt=1)

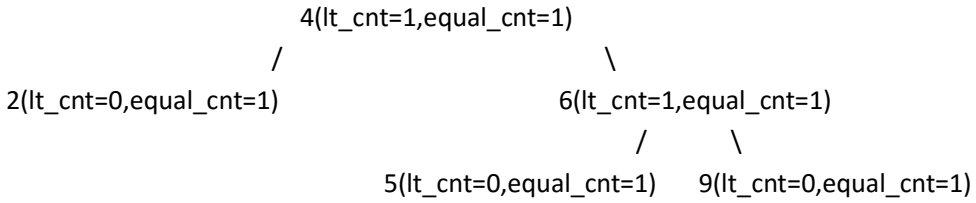
/

5(lt_cnt=0,equal_cnt=1)

---

ret = 4(lt_cnt=1,equal_cnt=1) = 2 （从根节点往右走才计算）

5)插入 9



---

ret = 4(lt_cnt=1,equal_cnt=1) + 6(lt_cnt=1,equal_cnt=1) =4 （从根节点往右走才计算）

节点数据结构定义：

```
47  struct Node
48  {
49      int val;
50      Node *left,*right;
51      int lt_cnt;
52      int equal_cnt;
53      Node(int val)
54      {
55          this->val = val;
56          left = right = NULL;
57          lt_cnt = 0;
58          equal_cnt = 1;
59      }
60  };
```

```
class BST
{
public:
    Node * root;
    BST()
    {
        root = NULL;
    }

    int addNode(int val)
    {
        if (!root)
        {
            root = new  Node(val);
            return 0;
        }
        Node* curNode = root;
        int cnt = 0;
        while (curNode)
        {
            if (val > curNode->val)//greater than
            {
                cnt += (curNode->equal_cnt + curNode->lt_cnt);//注意1：只有往右走才计算小于等于的值
                if (curNode->right == NULL)
                {
                    curNode->right = new    Node(val);//注意:new一个节点时，小于新节点的节点数肯定为0，因此不用累计
                    break;
                }
                curNode = curNode->right;
            }
            else if (val < curNode->val)//less than
            {
                curNode->lt_cnt += 1;
                if (curNode->left == NULL)
                {
                    curNode->left = new Node(val);
                    break;
                }
                curNode = curNode->left;
            }
            else//equal
            {
                cnt += curNode->lt_cnt;//注意2：加上小于当前节点值的节点数
                curNode->equal_cnt += 1;
                break;
            }
        }
        return cnt;
    }
```

主函数：

```
class Solution {
public:
    vector<int> countSmaller(vector<int>& nums) {
        vector<int>res(nums.size(),0);
        int i = nums.size()-1;
        BST bst = BST();
        for(auto p = nums.rbegin();p!=nums.rend();p++)//注意倒序插入
        {
            res[i--] = bst.addNode(*p);
        }
        return res;
    }
};
```

---

316. Remove Duplicate Letters

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

Given "bcabc"

Return "abc"

Given "cbacdcbc"

Return "acdb"

思路：
1)用 used 标志位标志当前这个字符有没有出现在栈里(相当于一个 map)
2)维护一个当前最小字母序的 stack，当前字符加入的时候，移除栈顶所有能移除(??)的比它大的字符————类似一种贪心策略，小字母序的能往前放就尽量往前放
所谓能移除是指：比它大的字符在后面还会出现(如何知道？对每个字符设置一个计数器)
证明：因为对当前字符而言，通过以上策略当前字符最后放的位置已经是最能靠前的位置
3)由 2 可知,栈始终维持最小字母序，因此如果当前字符在栈中已经存在，可以直接跳过

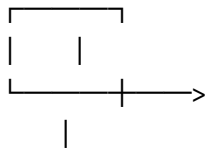
```cpp
class Solution {
public:
    string removeDuplicateLetters(string s) {
        int cnt[26] = { 0 };//剩余字符计数器
        unsigned used = 0;//used标识，用低26位标识当前字符是否已经在栈中
        for (auto c : s)//初始化计数器
            cnt[c - 'a'] += 1;
        stack<char> res_stack;
        for (auto c : s)
        {
            cnt[c - 'a']--;//处理到当前字符，字符数减1
            if(res_stack.empty())//栈为空可以直接加入
            {
               res_stack.push(c);
               used |= (1 << (c - 'a'));//修改used标志位
            }
            else if((used>>(c-'a')&1)==1)//已经在栈中出现的字符直接跳过
                continue;
            else
            {//当前字符没出现过，弹出所有比当前字符字母序大的字符，并修改used标志位；然后压入当前字符，修改标志位
                while (!res_stack.empty() && c<res_stack.top() && cnt[res_stack.top() - 'a']>0)
                {
                    used &= (~(1 << (res_stack.top() - 'a')));
                    res_stack.pop();
                }

                res_stack.push(c);
                used |= (1 << (c - 'a'));
            }
        }
        string res = "";
        while (!res_stack.empty())
        {
            res = res_stack.top() + res;
            res_stack.pop();
        }
        return res;
    }
};
```

---

335. Self Crossing My Submissions QuestionEditorial Solution
Total Accepted: 4810 Total Submissions: 25773 Difficulty: Hard
You are given an array x of n positive numbers. You start at point (0,0) and moves x[0] metres to the north, then x[1] metres to the west, x[2] metres to the south, x[3] metres to the east and so on. In other words, after each move your direction changes counter-clockwise.

Write a one-pass algorithm with O(1) extra space to determine, if your path crosses itself, or not.

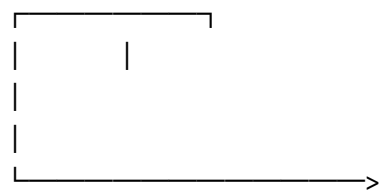
Example 1:
Given x = [2, 1, 1, 2],



Return true (self crossing)

Example 2:

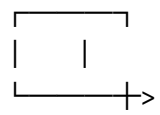
Given x = [1, 2, 3, 4],



Return false (not self crossing)

Example 3:

Given x = [1, 1, 1, 1],



Return true (self crossing)

思路：

1）总结出相交的模式，然后依次轮流判断是否满足

1 2 3 4 5 2 8 7

->判断 1 2 3 4 5 2 不满足模式

->判断 2 3 4 5 2 8 不满足模式

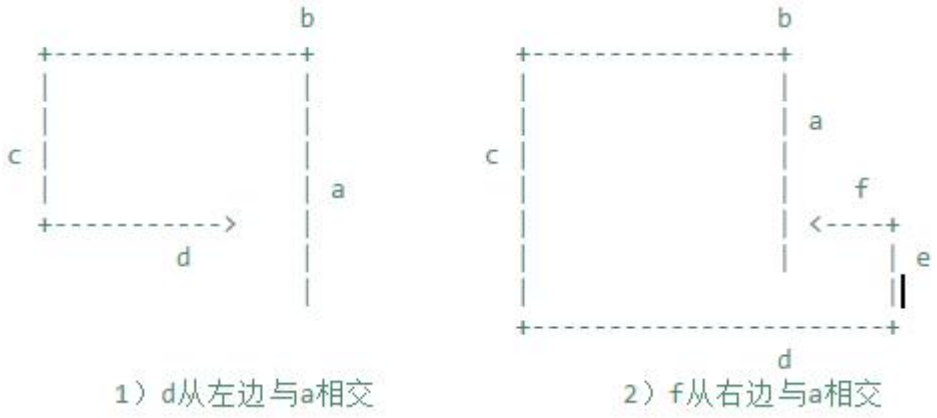
->判断 3 4 5 2 8 7 不满足模式

->判断 4 5 2 8 7 0 不满足模式

->判断 5 2 8 7 0 0 不满足模式

....

2)每 6 个数为一组表示一个模式:



相交分为以上 2 种情况:

1)第一种情况需要 4 条线，第二种情况需要 6 跳线；为统一起见，6 个数判定一次，4 跳线相交的模式可以认为 e=f=0(补 0)

2)a,b,c 三条线不管怎么都不会相交,因此约束条件从第 4 跳线 d 开始

3)以上两种情形都要求:d>=b>0(注意 b=0 说明是补的 0，这时不能认为是模式),为保证 d 能交上 a:a>=c ;为保证 f 能交上 a: a>=c-e and f>=d-b

```
class Solution {
public:
    bool isSelfCrossing(vector<int>& x)
    {
        int a = 0, b = 0, c = 0, d = 0, e = 0, f = 0;
        for (int i=x.size()-1;i>=0;i--) //逆序判定（为了便于实现，正序也可）
        {
            f = e; e = d; d = c; c = b; b = a; a = x[i];
            if(ValidPattern(a,b,c,d,e,f))
                return true;
        }
        return false;
    }

    bool ValidPattern(int a,int b,int c,int d,int e,int f)//相交模式
    {
        return (d>=b&&b>0) &&( (a>=c&&c>0) or ((a>=c-e)&&(c-e>0)  && (f>=d-b)&&(d-b>=0) ));
    }
};
```