LeetCode 307

https://leetcode.com/problems/range-sum-query-mutable/description/

Yifeng Zeng

Description

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

Idea Report

We are asked to get the sum of the elments in a range, so my primitive idea is to use a special data structure called prefix sum array, int[] prefixSum. For input array nums, prefixSum[i] means the sum of first ith elements in nums array, and prefixSum[0] is 0. For example, for input nums = [1,3,5] then prefixSum is [0,1,4,9]. prefixSum[0] is basically 0, prefixSum[1] is the first element which is 1, prefixSum[2] is the sum of first 2 elements, which is 1 + 3 = 4, and prefixSum[3] is the sum of first 3 elements, which is 1 + 3 + 5 = 9. When query the sum ranging from i to j we can just return prefixSum[j + 1] - prefixSum[i] which is sum of index [0, 1, 2,…,i,…,j] - sum of index [0, 1, 2, …,i-1] which is exactly sum of index [i,i+1,…j-1,j]. To update the prefixSum at index i to value val, we can just find out all the prefix sums that contains the index i number, substract the old value

and plus the new value. The old value can be obtains by query(i, i). To build the prefix sum array, uses O(n) time, to query(i, j) uses O(1) time, to update val at index i uses O(n - i) time which is essentially O(n) time.

Code

```
class NumArray {
    // TLE
    int[] prefixSum;
    public NumArray(int[] nums) {
        prefixSum = new int[nums.length + 1];
        for (int i = 1; i < prefixSum.length; i++) {
            prefixSum[i] = prefixSum[i - 1] + nums[i - 1];
        }
    }

    public void update(int i, int val) {
        int oldValue = sumRange(i, i);
        for (int j = i + 1; j < prefixSum.length; j++) {
            prefixSum[j] = prefixSum[j] - oldValue + val;
        }
    }

    public int sumRange(int i, int j) {
        return prefixSum[j + 1] - prefixSum[i];
    }
}
```

The prefix sum method takes O(n) time to update which is relatively time consuming when the input size is large. And the problems states that "calls to update and sumRange function is distributed evenly." so we need a better approach than O(n), which is O(logN). So we can think of a tree structure, an advanced data structure is called segment tree. Take the same input example [1,3,4], we can draw a tree like follows:

```
              [0, 2 : 8]
             /           \
      [0, 1 : 4]          [2, 2 : 4]
      /        \
[0, 0 : 0]     [1, 1 : 3]
```

Each node has three integer and to children. The first integer is start, meaning the start index of the array. The second integer is end, meaning the end index of the array. The third integer is sum,

meaning teh sum of elements between start and end inclusive. So for each leave node, the sum is the element value at that index, for example [1,1:3] is the node indicates index start from 1 end at 1 which is nums[1], so the sum is nums[1] = 3. For other nodes, the sum is a range, for example [0,1:4] is the node indexates index start from 0 end at 1 which is sum of nums[0], nums[1], which is 4. To query, we can check if the query index i, j matches node.start, node.end, if it matches we just return node.sum; if it doesn't match, we can split the indexes into two parts and find the result either from left child or right child or both. For example, if we want to query [0,2], it matches root we can just return 8 directly. If we want to query [1,2], we see node[0,2:8] doesn't match, then we devide query into two halves, to query [1,1] from left child, and [2,2] from right child (which we get [2,2:4] directly). From left child[0,1:4], the indexes doesn't match [1,1], so we split again to get from [1,1:3]. And we sum together 3 + 4 together to get query[1,2] = 7. To update the value at index i, we can find the [i,i:value] node, and update all nodes along the path from root to node [i,i:value] by substracting the old value and plus the new value.

Code

```
class NumArray {

    SegmentTreeNode root;

    public NumArray(int[] nums) {
        root = build(nums, 0, nums.length - 1);
        // print(root);
    }

    public void update(int i, int val) {
        // print(root);
        modify(root, i, val);
        // print(root);
    }

    public int sumRange(int i, int j) {
        return query(root, i, j);
    }

    class SegmentTreeNode {
        int start;
        int end;
        int sum;
        SegmentTreeNode left;
        SegmentTreeNode right;
        public SegmentTreeNode(int start, int end, int sum) {
            this.start = start;
            this.end = end;
            this.sum = sum;
            this.left = null;
```

```
            this.right = null;
        }
    }

    private SegmentTreeNode build(int[] arr, int start, int end) {
        if (start > end) {
            return null;
        }
        if (start == end) {
            return new SegmentTreeNode(start, end, arr[start]);
        }

        int mid = (end - start) / 2 + start;
        SegmentTreeNode node = new SegmentTreeNode(start, end, 0);
        node.left = build(arr, start, mid);
        node.right = build(arr, mid + 1, end);
        if (node.left != null) {
            node.sum += node.left.sum;
        }
        if (node.right != null) {
            node.sum += node.right.sum;
        }
        return node;
    }

    private int query(SegmentTreeNode cur, int start, int end) {
        if (cur == null || start > end) {
            return 0;
        }
        if (start <= cur.start && cur.end <= end) {
            return cur.sum;
        }

        int mid = (cur.end - cur.start) / 2 + cur.start;
        if (end <= mid) {
            return query(cur.left, start, end);
        } else if (start > mid) {
            return query(cur.right, start, end);
        }
        return query(cur.left, start, mid) + query(cur.right, mid + 1, end);
    }

    private int modify(SegmentTreeNode cur, int i, int val) {
        if (cur == null) {
            return 0;
        }
        if (i == cur.start && i == cur.end) {
            int oldValue = cur.sum;
            cur.sum = val;
            return oldValue;
        }
```

```
            int mid = (cur.end - cur.start) / 2 + cur.start;
            int oldValue = 0;
            if (cur.start <= i && i <= mid) {
                oldValue = modify(cur.left, i, val);
            } else if (mid + 1 <= i && i <= cur.end) {
                oldValue = modify(cur.right, i, val);
            }
            cur.sum = cur.sum - oldValue + val;
            return oldValue;
        }

        // private void print(SegmentTreeNode root) {
        //     if (root == null) {
        //         return;
        //     }
        //
        //     System.out.println(root.start + "," + root.end + ", " + root.sum);
        //     print(root.left);
        //     print(root.right);
        // }
    }
```

Summary

- Prefix sum takes O(1) to get the sum of a range in the array, which would be a good choice if the array is immutable.
- Segment tree takes O(n) time to build and O(logN) time to query and update.