307. Range Sum Query - Mutable

April 4, 2016 | 70.3K views

Average Rating: 4.75 (101 votes)

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* ($i \le j$), 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:

1. The array is only modifiable by the *update* function.
2. You may assume the number of calls to *update* and *sumRange* function is distributed evenly.

Summary

This article is for intermediate level readers. It introduces the following concepts: Range sum query, Sqrt decomposition, Segment tree.

Solution

Approach 1: Naive

Algorithm

A trivial solution for Range Sum Query - `RSQ(i, j)` is to iterate the array from index i to j and sum each element.

Java

```java
private int[] nums;
public int sumRange(int i, int j) {
    int sum = 0;
    for (int l = i; l <= j; l++) {
        sum += data[l];
    }
    return sum;
}

public int update(int i, int val) {
    nums[i] = val;
}
// Time Limit Exceeded
```

Complexity Analysis

- Time complexity : $O(n)$ - range sum query, $O(1)$ - update query

  For range sum query we access each element from the array for constant time and in the worst case we access n elements. Therefore time complexity is $O(n)$. Time complexity of update query is $O(1)$.

- Space complexity : $O(1)$.

Approach 2: Sqrt Decomposition

Intuition

The idea is to split the array in blocks with length of $\sqrt{n}$. Then we calculate the sum of each block and store it in auxiliary memory `b`. To query `RSQ(i, j)`, we will add the sums of all the blocks lying inside and those that partially overlap with range $[i \ldots j]$.

Algorithm

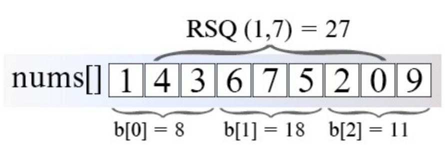

Figure 1. Range sum query using SQRT decomposition.

In the example above, the array `nums`'s length is `9`, which is split into blocks of size $\sqrt{9}$. To get `RSQ(1, 7)` we add `b[1]`. It stores the sum of `range [3, 5]` and partially sums from `block 0` and `block 2`, which are overlapping boundary blocks.

Java

```java
        b[i / len] += nums[i];
}

public int sumRange(int i, int j) {
    int sum = 0;
    int startBlock = i / len;
    int endBlock = j / len;
    if (startBlock == endBlock) {
        for (int k = i; k <= j; k++)
            sum += nums[k];
    } else {
        for (int k = i; k <= (startBlock + 1) * len - 1; k++)
            sum += nums[k];
        for (int k = startBlock + 1; k <= endBlock - 1; k++)
            sum += b[k];
        for (int k = endBlock * len; k <= j; k++)
            sum += nums[k];
    }
    return sum;
}

public void update(int i, int val) {
    int b_l = i / len;
    b[b_l] = b[b_l] - nums[i] + val;
    nums[i] = val;
}
// Accepted
```

Complexity Analysis

- Time complexity : $O(n)$ - preprocessing, $O(\sqrt{n})$ - range sum query, $O(1)$ - update query

  For range sum query in the worst-case scenario we have to sum approximately $3\sqrt{n}$ elements. In this case the range includes $\sqrt{n}-2$ blocks, which total sum costs $\sqrt{n}-2$ operations. In addition to this we have to add the sum of the two boundary blocks. This takes another $2(\sqrt{n}-1)$ operations. The total amount of operations is around $3\sqrt{n}$.

- Space complexity : $O(\sqrt{n})$.

  We need additional $\sqrt{n}$ memory to store all block sums.

Approach 3: Segment Tree

Algorithm

Segment tree is a very flexible data structure, because it is used to solve numerous range query problems like finding minimum, maximum, sum, greatest common divisor, least common denominator in array in logarithmic time.

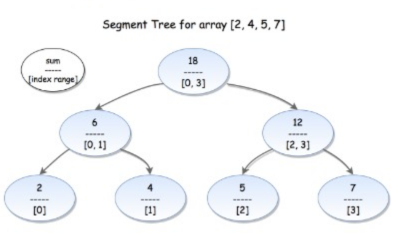

Figure 2. Illustration of Segment tree.

The segment tree for array $a[0, 1, \ldots, n-1]$ is a binary tree in which each node contains **aggregate** information (min, max, sum, etc.) for a subrange $[i \ldots j]$ of the array, as its left and right child hold information for range $[i \ldots \frac{i+j}{2}]$ and $[\frac{i+j}{2}+1, j]$.

Segment tree could be implemented using either an array or a tree. For an array implementation, if the element at index i is not a leaf, its left and right child are stored at index $2i$ and $2i+1$ respectively.

In the example above (Figure 2), every leaf node contains the initial array elements `{2,4,5,7}`. The internal nodes contain the sum of the corresponding elements in range, *e.g.* `(6)` is the sum for the elements from index 0 to index 1. The root `(18)` is the sum of its children `(6)` and `(12)`, which also holds the total sum of the entire array.

Segment Tree can be broken down to the three following steps:

1. Pre-processing step which builds the segment tree from a given array.
2. Update the segment tree when an element is modified.
3. Calculate the Range Sum Query using the segment tree.

1. Build segment tree

We will use a very effective bottom-up approach to build segment tree. We already know from the above that if some node p holds the sum of $[i \ldots j]$ range, its left and right children hold the sum for range $[i \ldots \frac{i+j}{2}]$ and $[\frac{i+j}{2}+1, j]$ respectively.

Therefore to find the sum of node p, we need to calculate the sum of its right and left child in advance.

We begin from the leaves, initialize them with input array elements $a[0, 1, \ldots, n-1]$. Then we move upward to the higher level to calculate the parents' sum till we get to the root of the segment tree.

Java

```java
int[] tree;
int n;
public NumArray(int[] nums) {
    if (nums.length > 0) {
        n = nums.length;
        tree = new int[n * 2];
        buildTree(nums);
    }
}
private void buildTree(int[] nums) {
    for (int i = n, j = 0;  i < 2 * n; i++,  j++)
        tree[i] = nums[j];
    for (int i = n - 1; i > 0; --i)
        tree[i] = tree[i * 2] + tree[i * 2 + 1];
}
```

Complexity Analysis

- Time complexity : $O(n)$

  Time complexity is $O(n)$, because we calculate the sum of one node during each iteration of the for loop. There are approximately $2n$ nodes in a segment tree.

  This could be proved in the following way: Segmented tree for array with n elements has n leaves (the array elements itself). The number of nodes in each level is half the number in the level below.

  So if we sum the number by level we will get:

  $$n + n/2 + n/4 + n/8 + \ldots + 1 \approx 2n$$

- Space complexity : $O(n)$.

  We used $2n$ extra space to store the segment tree.

2. Update segment tree

When we update the array at some index i we need to rebuild the segment tree, because there are tree nodes which contain the sum of the modified element. Again we will use a bottom-up approach. We update the leaf node that stores $a[i]$. From there we will follow the path up to the root updating the value of each parent as a sum of its children values.

Java

```java
void update(int pos, int val) {
    pos += n;
    tree[pos] = val;
    while (pos > 0) {
        int left = pos;
        int right = pos;
        if (pos % 2 == 0) {
            right = pos + 1;
        } else {
            left = pos - 1;
        }
        // parent is updated after child is updated
        tree[pos / 2] = tree[left] + tree[right];
        pos /= 2;
    }
}
```

Complexity Analysis

- Time complexity : $O(\log n)$.

  Algorithm has $O(\log n)$ time complexity, because there are a few tree nodes with range that include ith array element, one on each level. There are $\log(n)$ levels.

- Space complexity : $O(1)$.

3. Range Sum Query

We can find range sum query $[L, R]$ using segment tree in the following way:

Algorithm hold loop invariant:

$l \le r$ and sum of $[L \ldots l]$ and $[r \ldots R]$ has been calculated, where l and r are the left and right boundary of calculated sum. Initially we set l with left leaf L and r with right leaf R. Range $[l, r]$ shrinks on each iteration till range borders meets after approximately $\log n$ iterations of the algorithm

- Loop till $l \le r$
  - Check if l is right child of its parent P
    - l is right child of P. Then P contains sum of range of l and another child which is outside the range $[l, r]$ and we don't need parent P sum. Add l to sum without its parent P and set l to point to the right of P on the upper level.
    - l is not right child of P. Then parent P contains sum of range which lies in $[l, r]$. Add P to sum and set l to point to the parent of P
  - Check if r is left child of its parent P
    - r is left child of P. Then P contains sum of range of r and another child which is outside the range $[l, r]$ and we don't need parent P sum. Add r to sum without its parent P and set r to point to the left of P on the upper level.
    - r is not left child of P. Then parent P contains sum of range which lies in $[l, r]$. Add P to sum and set r to point to the parent of P

Java

```java
public int sumRange(int l, int r) {
    // get leaf with value 'l'
    l += n;
    // get leaf with value 'r'
    r += n;
    int sum = 0;
    while (l <= r) {
        if ((l % 2) == 1) {
           sum += tree[l];
           l++;
        }
        if ((r % 2) == 0) {
           sum += tree[r];
           r--;
        }
        l /= 2;
        r /= 2;
    }
    return sum;
}
```

Complexity Analysis

- Time complexity : $O(\log n)$

  Time complexity is $O(\log n)$ because on each iteration of the algorithm we move one level up, either to the parent of the current node or to the next sibling of parent to the left or right direction till the two boundaries meet. In the worst-case scenario this happens at the root after $\log n$ iterations of the algorithm.

- Space complexity : $O(1)$.

Further Thoughts

The iterative version of Segment Trees was introduced in this article. A more intuitive, recursive version of Segment Trees to solve this problem is discussed here. The concept of Lazy Propagation is also introduced there.

There is an alternative solution of the problem using Binary Indexed Tree. It is faster and simpler to code. You can find it here.