Previous Next

303. Range Sum Query - Immutable

March 5, 2016 | 74.7K views

★★★★★
Average Rating: 4.69 (101 votes)

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

Example:

```
Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

Note:

1. You may assume that the array does not change.
2. There are many calls to *sumRange* function.

Solution

Approach #1 (Brute Force) [Time Limit Exceeded]

Each time *sumRange* is called, we use a for loop to sum each individual element from index i to j.

```java
private int[] data;

public NumArray(int[] nums) {
    data = nums;
}

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

Complexity analysis:

- Time complexity : $O(n)$ time per query. Each *sumRange* query takes $O(n)$ time.
- Space complexity : $O(1)$. Note that `data` is a *reference* to `nums` and is not a copy of it.

Approach #2 (Caching) [Accepted]

Imagine that *sumRange* is called one thousand times with the exact same arguments. How could we speed that up?

We could trade in extra space for speed. By pre-computing all range sum possibilities and store its results in a hash table, we can speed up the query to constant time.

```java
private Map<Pair<Integer, Integer>, Integer> map = new HashMap<>();

public NumArray(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        int sum = 0;
        for (int j = i; j < nums.length; j++) {
            sum += nums[j];
            map.put(Pair.create(i, j), sum);
        }
    }
}

public int sumRange(int i, int j) {
    return map.get(Pair.create(i, j));
}
```

Complexity analysis

- Time complexity : $O(1)$ time per query, $O(n^2)$ time pre-computation. The pre-computation done in the constructor takes $O(n^2)$ time. Each *sumRange* query's time complexity is $O(1)$ as the hash table's look up operation is constant time.
- Space complexity : $O(n^2)$. The extra space required is $O(n^2)$ as there are n candidates for both i and j.

Approach #3 (Caching) [Accepted]

The above approach takes a lot of space, could we optimize it?

Imagine that we pre-compute the cummulative sum from index 0 to k. Could we use this information to derive $Sum(i, j)$?

Let us define $sum[k]$ as the cumulative sum for $nums[0 \cdots k-1]$ (inclusive):

$$sum[k] = \begin{cases} \sum_{i=0}^{k-1} nums[i] & , k > 0 \\ 0 & , k = 0 \end{cases}$$

Now, we can calculate *sumRange* as following:

$$sumRange(i, j) = sum[j+1] - sum[i]$$

```java
private int[] sum;

public NumArray(int[] nums) {
    sum = new int[nums.length + 1];
    for (int i = 0; i < nums.length; i++) {
        sum[i + 1] = sum[i] + nums[i];
    }
}

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

Notice in the code above we inserted a dummy 0 as the first element in the *sum* array. This trick saves us from an extra conditional check in *sumRange* function.

Complexity analysis

- Time complexity : $O(1)$ time per query, $O(n)$ time pre-computation. Since the cumulative sum is cached, each *sumRange* query can be calculated in $O(1)$ time.
- Space complexity : $O(n)$.

Rate this article: ★★★★★

Previous Next

Comments: 43

Sort By

Type comment here... (Markdown is supported)

Preview Post

pgalatic ★ 22 November 1, 2018 7:33 PM

I did the brute force version, which was still accepted, despite supposedly exceeding the time limit.

22 Share Reply

SHOW 3 REPLIES

terrible_whiteboard ★ 633 May 19, 2020 6:27 PM

I made a video if anyone is having trouble understanding the solution (clickable link)
https://youtu.be/CjPMfq3ULZg

Read More

20 Share Reply

SHOW 2 REPLIES

Yurlungur ★ 16 June 29, 2019 10:27 AM

Approach 2 is rejected in python. Approach 3 is really cool and elegant.

14 Share Reply

SHOW 1 REPLY

Guangyiz ★ 4 December 29, 2017 2:10 PM

i use two-dimensional array instead of pair,but the result is memory limit exceeded.

4 Share Reply

SHOW 3 REPLIES

NideeshT ★ 590 April 30, 2019 5:49 AM

Java Code + Youtube Video Explanation - accepted https://youtu.be/oLW47T7WIUw (clickable link)

Read More

3 Share Reply

Cronek ★ 91 March 16, 2018 9:54 PM

It is true you can get overflow in the third solution, but this doesn't invalidate the method.
Let's try an example with an array of 300 entries of 100 million. The sum will obviously overflow, but when calculating sumRange(i, j), we can distinguish two possibilities:

- the sum doesn't overflow in the interval: sum[i] = actual sum up to i - x * 2^32-1 and

Read More

3 Share Reply

yarramsetti ★ 2 May 21, 2019 9:50 PM

Store the sum back in the input array that elimates the need for extra space

3 Share Reply

xin_w ★ 2 March 4, 2019 3:22 PM

Implementation with c++.

```
class NumArray {
public:
    NumArray(vector<int> nums) {
```

Read More

1 Share Reply

yinjiecheng ★ 42 December 24, 2016 10:40 PM

#3 is so brilliant!

1 Share Reply

jfyh5388 ★ 0 August 22, 2016 5:48 PM

When I run the The Approach #2,the result is :"Compile Error.Line 2: error: cannot find symbol: class pair".
And I implemented the Approach #2 in C++,the result is:"Time Limit Exceeded."And the last executed input is a very large number.I want to know why is it?

1 Share Reply

SHOW 1 REPLY

‹ 1 2 3 4 5 ›