Solution

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

**Algorithm**

Each time *sumRegion* is called, we use a double for loop to sum all elements from (row1, col1) \rightarrow (row2, col2)(*row*1,*col*1)→(*row*2,*col*2).

private int[][] data;

public NumMatrix(int[][] matrix) {

data = matrix;

}

public int sumRegion(int row1, int col1, int row2, int col2) {

int sum = 0;

for (int r = row1; r <= row2; r++) {

for (int c = col1; c <= col2; c++) {

sum += data[r][c];

}

}

return sum;

}

**Complexity analysis**

* Time complexity : O(mn)*O*(*mn*) time per query. Assume that m*m* and n*n* represents the number of rows and columns respectively, each *sumRegion* query can go through at most m \times n*m*×*n* elements.
* Space complexity : O(1)*O*(1). Note that data is a *reference* to matrix and is not a copy of it.

Approach #2 (Caching) [Memory Limit Exceeded]

**Intuition**

Since *sumRegion* could be called many times, we definitely need to do some pre-processing.

**Algorithm**

We could trade in extra space for speed by pre-calculating all possible rectangular region sum and store them in a hash table. Each *sumRegion* query now takes only constant time complexity.

**Complexity analysis**

* Time complexity : O(1)*O*(1) time per query, O(m^2n^2)*O*(*m*2*n*2) time pre-computation. Each *sumRegion* query takes O(1)*O*(1) time as the hash table lookup's time complexity is constant. The pre-computation will take O(m^2n^2)*O*(*m*2*n*2) time as there are a total of m^2 \times n^2*m*2×*n*2 possibilities need to be cached.
* Space complexity : O(m^2n^2)*O*(*m*2*n*2). Since there are mn*mn* different possibilities for both top left and bottom right points of the rectangular region, the extra space required is O(m^2n^2)*O*(*m*2*n*2).

Approach #3 (Caching Rows) [Accepted]

**Intuition**

Remember from the [1D version](https://leetcode.com/course/chapters/leetcode-101/range-sum-query-immutable/) where we used a cumulative sum array? Could we apply that directly to solve this 2D version?

**Algorithm**

Try to see the 2D matrix as m*m* rows of 1D arrays. To find the region sum, we just accumulate the sum in the region row by row.

private int[][] dp;

public NumMatrix(int[][] matrix) {

if (matrix.length == 0 || matrix[0].length == 0) return;

dp = new int[matrix.length][matrix[0].length + 1];

for (int r = 0; r < matrix.length; r++) {

for (int c = 0; c < matrix[0].length; c++) {

dp[r][c + 1] = dp[r][c] + matrix[r][c];

}

}

}

public int sumRegion(int row1, int col1, int row2, int col2) {

int sum = 0;

for (int row = row1; row <= row2; row++) {

sum += dp[row][col2 + 1] - dp[row][col1];

}

return sum;

}

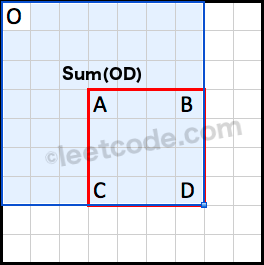
**Complexity analysis**

* Time complexity : O(m)*O*(*m*) time per query, O(mn)*O*(*mn*) time pre-computation. The pre-computation in the constructor takes O(mn)*O*(*mn*) time. The *sumRegion* query takes O(m)*O*(*m*) time.
* Space complexity : O(mn)*O*(*mn*). The algorithm uses O(mn)*O*(*mn*) space to store the cumulative sum of all rows.

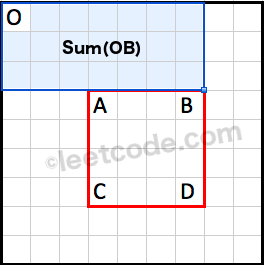
Approach #4 (Caching Smarter) [Accepted]

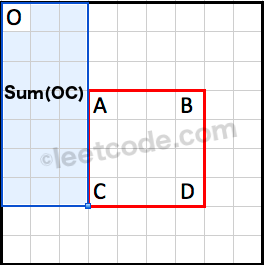
**Algorithm**

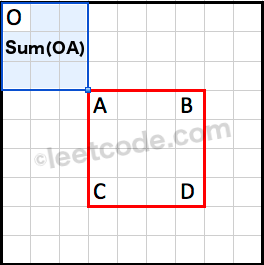
We used a cumulative sum array in the [1D version](https://leetcode.com/course/chapters/leetcode-101/range-sum-query-immutable/). We notice that the cumulative sum is computed with respect to the origin at index 0. Extending this analogy to the 2D case, we could pre-compute a cumulative region sum with respect to the origin at (0, 0)(0,0).

  
Sum(OD) is the cumulative region sum with respect to the origin at (0, 0).

How do we derive Sum(ABCD)*Sum*(*ABCD*) using the pre-computed cumulative region sum?

  
Sum(OB) is the cumulative region sum on top of the rectangle.

  
Sum(OC) is the cumulative region sum to the left of the rectangle.

  
Sum(OA) is the cumulative region sum to the top left corner of the rectangle.

Note that the region Sum(OA)*Sum*(*OA*) is covered twice by both Sum(OB)*Sum*(*OB*) and Sum(OC)*Sum*(*OC*). We could use the principle of inclusion-exclusion to calculate Sum(ABCD)*Sum*(*ABCD*) as following:

Sum(ABCD) = Sum(OD) - Sum(OB) - Sum(OC) + Sum(OA)*Sum*(*ABCD*)=*Sum*(*OD*)−*Sum*(*OB*)−*Sum*(*OC*)+*Sum*(*OA*)

private int[][] dp;

public NumMatrix(int[][] matrix) {

if (matrix.length == 0 || matrix[0].length == 0) return;

dp = new int[matrix.length + 1][matrix[0].length + 1];

for (int r = 0; r < matrix.length; r++) {

for (int c = 0; c < matrix[0].length; c++) {

dp[r + 1][c + 1] = dp[r + 1][c] + dp[r][c + 1] + matrix[r][c] - dp[r][c];

}

}

}

public int sumRegion(int row1, int col1, int row2, int col2) {

return dp[row2 + 1][col2 + 1] - dp[row1][col2 + 1] - dp[row2 + 1][col1] + dp[row1][col1];

}

**Complexity analysis**

* Time complexity : O(1)*O*(1) time per query, O(mn)*O*(*mn*) time pre-computation. The pre-computation in the constructor takes O(mn)*O*(*mn*) time. Each *sumRegion* query takes O(1)*O*(1) time.
* Space complexity : O(mn)*O*(*mn*). The algorithm uses O(mn)*O*(*mn*) space to store the cumulative region sum.

[Report Article Issue](https://github.com/LeetCode-Feedback/LeetCode-Feedback/issues)

Comments: 59

BestMost VotesNewest to OldestOldest to Newest



Preview

Post

[](https://leetcode.com/lucky1801)

[lucky1801](https://leetcode.com/lucky1801)278

January 5, 2019 1:29 AM

Read More

Thank you for this. Every article should be like this - very clear and concise.

**227**

Reply

Share

Report

[](https://leetcode.com/novice87)

[novice87](https://leetcode.com/novice87)515

November 3, 2019 7:33 PM

Read More

How is this derived?

dp[r + 1][c + 1] = dp[r + 1][c] + dp[r][c + 1] + matrix[r][c] - dp[r][c];

**40**

Show 9 replies

Reply

Share

Report

[](https://leetcode.com/Sehara_)

[Sehara\_](https://leetcode.com/Sehara_)15

October 16, 2016 2:15 PM

Read More

the last approach is excellent :)

**12**

Reply

Share

Report

[](https://leetcode.com/jordan34)

[jordan34](https://leetcode.com/jordan34)A picture containing text, sign

Description automatically generated1525

August 25, 2020 7:47 PM

Read More

Great explanation, but just one clarification. The reason we add the Sum(0A) region in the example is since we subtracted it twice, by removing both Sum(OB) and Sum(OC) regions. Both of these regions overlap Sum(0A). We need to add it back once otherwise we are miscalculating.

**5**

Show 1 reply

Reply

Share

Report

[](https://leetcode.com/hsuankuai)

[hsuankuai](https://leetcode.com/hsuankuai)15

May 11, 2019 1:58 AM

Read More

Here is a problem that the given range may exceed the boundary of the matrix, which was not mentioned in the description.

**15**

Show 1 reply

Reply

Share

Report

[](https://leetcode.com/max_nuudelchin)

[max\_nuudelchin](https://leetcode.com/max_nuudelchin)8

July 12, 2019 1:23 PM

Read More

Actually we don't need additional array. By modifying the matrix array, we achieve 0(1) space complexity.

**8**

Show 3 replies

Reply

Share

Report

[](https://leetcode.com/codewithcs)

[codewithcs](https://leetcode.com/codewithcs)120

December 18, 2020 1:07 AM

Read More

In Approach 3, Creating matrix[0].length + 1 columns in dp[][] in not at all intuitive.  
Instead create dp[][] as exactly the same size as matrix[][], dp = new int[matrix.length][matrix[0].length];

What I mean is this:

int[][] dp ;

public NumMatrix(int[][] matrix) {

if(matrix.length == 0 || matrix[0].length == 0) return ;

dp = new int[matrix.length][matrix[0].length];

for(int i=0; i<matrix.length; i++){

dp[i][0] = matrix[i][0];

}

for(int i=0; i< matrix.length; i++){

for(int j=1; j< matrix[0].length; j++){

dp[i][j] = dp[i][j-1] + matrix[i][j];

}

}

}

public int sumRegion(int row1, int col1, int row2, int col2) {

int sum = 0;

for(int row = row1; row <= row2; row++){

if(col1 == 0){

sum += dp[row][col2];

} else {

sum += dp[row][col2] - dp[row][col1-1];

}

}

return sum;

}

**3**

Show 1 reply

Reply

Share

Report

[](https://leetcode.com/antoniofraga)

[antoniofraga](https://leetcode.com/antoniofraga)29

Last Edit: May 22, 2019 2:38 PM

Read More

If you're allowed to change the input you can do things in place in approach #4. The memory complexity would be O(1) as well. Time complexity per query would still be O(1) amortized.

**2**

Show 2 replies

Reply

Share

Report

[](https://leetcode.com/pss314)

[pss314](https://leetcode.com/pss314)1

July 13, 2021 3:40 PM

Read More

Good explanation for answering "forest queries" found at <https://usaco.guide/silver/prefix-sums-2/#2d-prefix-sums>

**1**

Reply

Share

Report

[](https://leetcode.com/Yinyu78)

[Yinyu78](https://leetcode.com/Yinyu78)0

Last Edit: May 12, 2021 5:33 PM

Read More

Approach [#4](https://leetcode.com/problems/median-of-two-sorted-arrays) - the 0th row and column of dp are fully filled by 0, which is not necessary. Using dp of size m x n is enough, though sumRegion would be a bit more complicated:

return row1 == 0 && col1 == 0 ? dp[row2][col2] :

row1 == 0 ? dp[row2][col2] - dp[row2][col1 - 1] :

col1 == 0 ? dp[row2][col2] - dp[row1 - 1][col2] :

dp[row2][col2] - dp[row1 - 1][col2] - dp[row2][col1 - 1] + dp[row1 - 1][c