1476. Subrectangle Queries

Array

Matrix

Problem Description

Design

Medium

In this problem, we are asked to implement a class called SubrectangleQueries which encapsulates a 2D rectangle grid of integers provided during instantiation. The class needs to support two operations:

1. updateSubrectangle(int row1, int col1, int row2, int col2, int newValue): This method allows updating all values

- within a specified subrectangle to a new given value. The subrectangle to be updated is defined by its upper left coordinate (row1, col1) and its bottom right coordinate (row2, col2).

 2. getValue(int row, int col): This method retrieves the current value at a specific coordinate (row, col) in the rectangle.
- These methods must efficiently reflect any updates made by updateSubrectangle when getValue is called.

The naive approach to solve the updateSubrectangle operation would be to iterate over every cell in the subrectangle and update

Intuition

subrectangle being updated is large.

To optimize this, we can use an approach where we track only the updates made rather than applying them immediately to the entire subrectangle. Whenever an update operation is performed, we record the details of the update—the coordinates of the

it to newValue. However, this could become inefficient when there are many updates before a call to getValue, especially if the

subrectangle and the new value—in a list of operations, ops.

Then, when getValue is invoked for a specific cell, we iterate through the list of updates in reverse chronological order (latest operation first) because the most recent value is what we're interested in. We check if the queried cell falls within the

subrectangle of an update. If it does, we return the newValue from the first update operation that includes the cell. Otherwise, if no update operations include the cell, we return the original value of the cell from the initial rectangle grid.

Solution Approach

The solution for the SubrectangleQueries class leverages a key concept in programming known as lazy updating combined with

the use of a history list to save update operations. Let's break down the two primary methods provided by the solution.

When the class is initialized with a 2D array representing the rectangle, we store this array and initialize an empty list self.ops to record update operations:

self.ops = []

def getValue(self, row: int, col: int) -> int:

record update operations:

def __init__(self, rectangle: List[List[int]]):
 self.g = rectangle

```
The updateSubrectangle method doesn't modify the original grid immediately. Instead, it appends the update information as a tuple (row1, col1, row2, col2, newValue) to self.ops:

def updateSubrectangle(self, row1: int, col1: int, row2: int, col2: int, newValue: int) -> None: self.ops.append((row1, col1, row2, col2, newValue))
```

During the getValue method, we iterate backward through self.ops to check if the given row and col coordinates fall within any of the recorded subrectangles. If they do, it means that this was the last update that touched the cell before the getValue

```
for r1, c1, r2, c2, v in self.ops[::-1]:
    if r1 <= row <= r2 and c1 <= col <= c2:
        return v
return self.g[row][col]</pre>
```

request, and the newValue from that update is returned immediately without checking earlier updates:

operations.

In conclusion, the solution approach utilizes a history list mechanism to deftly manage multiple updates and retrieve operations without redundantly modifying the entire rectangle upon each update. This way, the process becomes markedly more efficient for scenarios involving many update operations and relatively few reads.

This approach is an application of the lazy evaluation pattern, as the updates to the grid are deferred and only evaluated when

needed. By applying this strategy, the algorithm ensures that unnecessary cell updates are avoided, reducing the number of

operations to be O(1) for each updateSubrectangle call, and O(k) for each getValue call, where k is the number of update

SubrectangleQueries class with the following 2D rectangle grid:

1 2 3
4 5 6
7 8 9

Let's use a small example to illustrate the solution approach for the SubrectangleQueries class. Suppose we initialize our

1. We perform an update on the subrectangle from (0, 0) to (1, 1) with a new value of 10. Our update call will be like this:

following:

The grid is instantiated as:

Example Walkthrough

This will not change the grid immediately but will record this operation in self.ops.

2. If we now call the getValue method to retrieve the value at (0, 1), which is part of the recently updated subrectangle, the method will do the

Since the most recent update included this cell with coordinates (0, 1), the method will return 10, which was the new value set

```
for that subrectangle.

3. Let's add another update, changing the value of the subrectangle from (1, 1) to (2, 2) to 20:
```

value = subrectangleQueries.getValue(2, 0)

self.grid = rectangle # Initialize the grid with the given rectangle

If there are no updates that affect the cell, return the original value

Remember that when using this code, you must also have the appropriate imports at the beginning of your script:

self.updates = [] # Keep a list to record all the updates made

self, row1: int, col1: int, row2: int, col2: int, newValue: int

Record the details of the update operation in the updates list

self.updates.append((row1, col1, row2, col2, newValue))

// Add the update operation to the list of operations

// Gets the current value of a cell after applying the updates

// Loop through the operations in reverse order

for (int $i = operations.size() - 1; i >= 0; --i) {$

int getValue(int row, int col) {

auto& op = operations[i];

return op[4];

operations.push_back({row1, col1, row2, col2, newValue});

// Check if the current cell is within the subrectangle bounds of a previous update

if (op[0] <= row && row <= op[2] && op[1] <= col && col <= op[3]) {</pre>

// If so, return the updated value for this cell

subrectangleQueries = SubrectangleQueries([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

subrectangleQueries.updateSubrectangle(0, 0, 1, 1, 10)

subrectangleQueries.updateSubrectangle(1, 1, 2, 2, 20)

Again, this updates the operation list but leaves the grid unchanged.

affecting it and thus returns the original value 7 from the original grid:

value = subrectangleQueries.getValue(0, 1)

value = subrectangleQueries.getValue(1, 1)

```
Throughout the example, we see that the updateSubrectangle method appends update operation details to the self.ops list but doesn't alter the original grid itself. When retrieving a value with getValue, the method checks the updates in reverse
```

returned. This optimization allows efficient handling of updates and retrievals by deferring actual updates until needed.

chronological order to see if they affect the cell in question. If they do, the latest value is returned. If not, the original grid value is

4. Another call to getValue for the coordinate (1, 1) would now return 20, as this is the newest value for that location due to the latest update.

5. If we ask for the value at (2, 0), which has not been touched by any update operations, the getValue method finds that there are no updates

Solution Implementation

Python

Iterate over the updates in reverse order (most recent first)
for r1, c1, r2, c2, value in reversed(self.updates):
 # If the cell (row, col) is within the updated subrectangle, return the new value
 if r1 <= row <= r2 and c1 <= col <= c2:
 return value</pre>

return self.grid[row][col]

class SubrectangleQueries:

) -> None:

def updateSubrectangle(

def __init__(self, rectangle: List[List[int]]):

def getValue(self, row: int, col: int) -> int:

```
# Example of how the SubrectangleQueries class is instantiated and used:
# obj = SubrectangleQueries(rectangle)
# obj.updateSubrectangle(row1, col1, row2, col2, newValue)
# param_2 = obj.getValue(row, col)
```
```

```python

Java

from typing import List

```
class SubrectangleQueries {
   private int[][] grid; // Matrix to represent the initial rectangle
   private LinkedList<int[]> updateOperations = new LinkedList<>(); // List to keep track of update operations
   // Constructor to initialize SubrectangleQueries with a rectangle
    public SubrectangleQueries(int[][] rectangle) {
       grid = rectangle;
   // Method to update a subrectangle.
   // (row1, col1) is the top left corner and (row2, col2) is the bottom right corner of the subrectangle.
   // newValue is the value to be updated in the subrectangle.
    public void updateSubrectangle(int row1, int col1, int row2, int col2, int newValue) {
       // Store the operation details at the beginning of the list for latest priority
       updateOperations.addFirst(new int[] { row1, col1, row2, col2, newValue });
   // Method to get the value of the cell at the specified row and column.
    public int getValue(int row, int col) {
       // Iterate over the operations in reverse order (start with the most recent one)
        for (int[] op : updateOperations) {
           // Check if the current cell was affected by the operation
           if (op[0] <= row && row <= op[2] && op[1] <= col && col <= op[3]) {</pre>
               return op[4]; // return the updated value if found
       // If no operations affected the cell, return the original value
       return grid[row][col];
/**
* The following is how you may instantiate and invoke methods of the SubrectangleQueries class:
 * SubrectangleQueries obj = new SubrectangleQueries(rectangle);
 * obj.updateSubrectangle(row1, col1, row2, col2, newValue);
* int val = obj.getValue(row, col);
*/
C++
#include <vector>
using namespace std;
// Class to handle subrectangle queries on a 2D array
class SubrectangleQueries {
private:
   vector<vector<int>> grid; // 2D vector to represent the initial rectangle
   vector<vector<int>> operations;
                                         // List of operations for updates
public:
   // Constructor that initializes the class with a rectangle
   SubrectangleQueries(vector<vector<int>>& rectangle) {
       grid = rectangle;
   // Updates the values of all cells in a subrectangle
   void updateSubrectangle(int row1, int col1, int row2, int col2, int newValue) {
```

```
// If no updates affected this cell, return the original value
        return grid[row][col];
};
/**
* How to use the class:
 * SubrectangleQueries* obj = new SubrectangleQueries(rectangle);
 * obj->updateSubrectangle(row1, col1, row2, col2, newValue);
 * int value = obj->getValue(row, col);
 * Note: You may wrap the usage within a main function if needed.
*/
TypeScript
// Define the rectangle grid and the operations log as global variables.
let rectangleGrid: number[][];
let opsLog: number[][];
// Initial setup for the rectangle grid.
function setupRectangle(rectangle: number[][]): void {
    rectangleGrid = rectangle;
    opsLog = [];
// Update a sub rectangle within the rectangle grid by logging the operation.
function updateSubrectangle(
    topLeftRow: number,
    topLeftCol: number,
    bottomRightRow: number,
    bottomRightCol: number,
    newValue: number,
): void {
    opsLog.push([topLeftRow, topLeftCol, bottomRightRow, bottomRightCol, newValue]);
// Get the current value of a cell in the rectangle grid, taking into account any updates.
function getValueAt(row: number, col: number): number {
    // Iterate through the operations log in reverse order to find the most recent update affecting the cell.
    for (let i = opsLog.length - 1; i >= 0; --i) {
        const [r1, c1, r2, c2, value] = opsLog[i];
       // Check if the cell lies within the bounds of the current operation.
        if (r1 <= row && row <= r2 && c1 <= col && col <= c2) {</pre>
            return value;
    // If no operations affect the cell, return the original value from the grid.
    return rectangleGrid[row][col];
// Example Usage:
// setupRectangle([[1, 2], [3, 4]]);
  updateSubrectangle(0, 0, 1, 1, 5);
// console.log(getValueAt(0, 0)); // Should output the updated value 5.
```

```
# obj.updateSubrectangle(row1, col1, row2, col2, newValue)
# param_2 = obj.getValue(row, col)

Remember that when using this code, you must also have the appropriate imports at the beginning of your script:
```

Example of how the SubrectangleQueries class is instantiated and used:

Time Complexity
 __init__(self, rectangle: List[List[int]]): This method initializes the object with the given rectangle. The time

Time and Space Complexity

class SubrectangleQueries:

) -> None:

python

from typing import List

def updateSubrectangle(

def __init__(self, rectangle: List[List[int]]):

def getValue(self, row: int, col: int) -> int:

return value

return self.grid[row][col]

obj = SubrectangleQueries(rectangle)

self.grid = rectangle # Initialize the grid with the given rectangle

self.updates = [] # Keep a list to record all the updates made

self, row1: int, col1: int, row2: int, col2: int, newValue: int

Record the details of the update operation in the updates list

Iterate over the updates in reverse order (most recent first)

If there are no updates that affect the cell, return the original value

If the cell (row, col) is within the updated subrectangle, return the new value

self.updates.append((row1, col1, row2, col2, newValue))

for r1, c1, r2, c2, value in reversed(self.updates):

if r1 <= row <= r2 and c1 <= col <= c2:</pre>

- complexity is 0(1) since it's simply storing the reference to rectangle and initializing an empty list ops.

 updateSubrectangle(self, row1: int, col1: int, row2: int, col2: int, newValue: int) -> None: This method records
- an update operation by appending a tuple to the ops list representing the subrectangle update parameters. The time complexity for each update is 0(1) because appending to a list in Python is an amortized constant time operation.
 getValue(self, row: int, col: int) -> int: This method retrieves the value of the cell at the specified row and column. It
- iterates over the ops list in reverse to find the most recent update that covers the cell in question. If k is the number of updates, the worst time complexity is O(k) because it might need to inspect every update in the worst case.

The space complexity for maintaining the rectangle is 0(m * n), where m is the number of rows and n is the number of columns in the given rectangle, since it stores the entire grid.

Space Complexity

columns in the given rectangle, since it stores the entire grid.

The space complexity for maintaining the ops list is O(u), where u is the number of update operations made. Each operation is stored as a tuple with five integers, so the total space taken by ops is proportional to the number of updates.