***Question 1***

1. **Initial Implementation:**

# Function to set a value at row x and column y in matrix A

def setValue(x, y, value):

A[x][y] = value

# Function to calculate the sum value of a rectangle enclosed within given points

def subRectangleSum(x1, y1, x2, y2):

total = 0

for i in range(x1, x2 + 1):

for j in range(y1, y2 + 1):

total += A[i][j]

return total

The time complexity of the **setValue()** function is constant or O(1) because it directly updates the value in the matrix.

The time complexity of the **subRectangleSum()** function is O((x2 - x1 + 1) \* (y2 - y1 + 1)) because it iterates over the sub-rectangle and sums up the values. This complexity is proportional to the size of the sub-rectangle.

1. **Optimized Implementation:**

To achieve a constant time complexity of O(1) for the **subRectangleSum()** function, we can use an extra matrix to pre-compute the cumulative sums of each sub-rectangle. This approach, known as the "prefix sum" or "integral image" technique, allows us to calculate the sum of any sub-rectangle in constant time.

Here's the modified implementation:

# Initialize the cumulative sum matrix

cumulative\_sum = [[0] \* N for \_ in range(N)]

# Function to set a value at row x and column y in matrix A

def setValue(x, y, value):

A[x][y] = value

updateCumulativeSum(x, y, value)

# Function to update the cumulative sum matrix

def updateCumulativeSum(x, y, value):

cumulative\_sum[x][y] = cumulative\_sum[x][y - 1] + cumulative\_sum[x - 1][y] - cumulative\_sum[x - 1][y - 1] + value

# Function to calculate the sum value of a rectangle enclosed within given points

def subRectangleSum(x1, y1, x2, y2):

if x1 == 0 and y1 == 0:

return cumulative\_sum[x2][y2]

elif x1 == 0:

return cumulative\_sum[x2][y2] - cumulative\_sum[x2][y1 - 1]

elif y1 == 0:

return cumulative\_sum[x2][y2] - cumulative\_sum[x1 - 1][y2]

else:

return cumulative\_sum[x2][y2] - cumulative\_sum[x2][y1 - 1] - cumulative\_sum[x1 - 1][y2] + cumulative\_sum[x1 - 1][y1 - 1]

In this optimized implementation, we maintain an additional matrix **cumulative\_sum**, where each cell **(i, j)** stores the cumulative sum of all elements in the sub-rectangle from **(0, 0)** to **(i, j)** in the original matrix **A**.

The **setValue()** function updates the value in the matrix **A** and calls the **updateCumulativeSum()** function to update the corresponding cumulative sum in the **cumulative\_sum** matrix. The **updateCumulativeSum()** function utilizes the prefix sum technique to efficiently calculate the cumulative sum

The **subRectangleSum()** function utilizes the cumulative sum matrix to calculate the sum of any sub-rectangle in constant time. It handles four cases based on the coordinates **(x1, y1)** and **(x2, y2)** to calculate the correct sum using the cumulative sums stored in the **cumulative\_sum** matrix.

The time complexity of the **setValue()** function is O(1) as it only involves updating the value in the matrix.

The time complexity of the **updateCumulativeSum()** function is O(1) as it only involves updating a constant number of cells in the **cumulative\_sum** matrix.

The time complexity of the **subRectangleSum()** function is O(1) as it directly computes the sum using the pre-computed cumulative sums. It does not depend on the size of the sub-rectangle, making it constant time.

.

Overall, this optimized implementation allows us to calculate the sum of any sub-rectangle in the matrix **A** in constant time, regardless of the size of the matrix or the sub-rectangle.

The initial implementation of the subRectangleSum() function iterates over each element in the specified sub-rectangle, resulting in a time complexity proportional to the sub-rectangle's size. To optimize it, we introduce an additional matrix called cumulative\_sum that stores the cumulative sums of each sub-rectangle in the original matrix A using the prefix sum technique. Whenever a value in A is updated, the corresponding value in cumulative\_sum is also updated. To calculate the sum of a sub-rectangle, we use the pre-computed cumulative sums, achieving a constant time complexity of O(1). This allows efficient handling of an indefinite stream of queries without the time complexity increasing with each query.

***Question 2:***

def balanced\_parenthesis(n):

"""

Creates a balanced parenthesis string of length n.

Args:

n: The length of the balanced parenthesis string.

Returns:

A balanced parenthesis string of length n.

"""

dp = [[0 for \_ in range(n + 1)] for \_ in range(n + 1)]

for i in range(n + 1):

for j in range(n + 1):

if i == 0 or j == 0:

dp[i][j] = 0

elif i == j:

dp[i][j] = 1

elif i < j:

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

# Construct the balanced parenthesis string.

string = ""

count = n

for i in range(2 \* n):

if i == 0:

string += "("

count -= 1

elif count > 0 and dp[count][i - 1] != dp[count - 1][i]:

string += "("

count -= 1

else:

string += ")"

return string

print(balanced\_parenthesis(3))

**Algorithm for Balanced Parentheses using Dynamic Programming :** Define a function **isBalanced(s)** that takes a string **s** as input and returns **true** if the parentheses in the string are balanced and **false** otherwise.

1. Create a 2D boolean array **dp** of size **(n+1) x (n+1)**, where **n** is the length of the input string **s**. This array will store the intermediate results of the subproblems.
2. Initialize the diagonal elements of **dp** to **true** since each individual parenthesis is balanced.
3. Iterate over the string length **len** from 2 to **n**:

* For **i** from 1 to **(n - len + 1)**, calculate the ending index **j** for the current substring starting at index **i** and having length **len**.
* Calculate the value of **dp[i][j]** as follows:
* Initialize **dp[i][j]** to **false**.
* For **k** from **i** to **j-1**, check if **dp[i][k]** and **dp[k+1][j]** are both **true**, and if **s[k]** and **s[j]** form a pair of parentheses (**'('** and **')'** or **'['** and **']'** or **'{'** and **'}'**).
* If both conditions are satisfied, set **dp[i][j]** to **true**.

5. Return **dp[1][n]**, where **n** is the length of the input string **s**. This will be **true** if the entire string has balanced parentheses, and **false** otherwise.

**Space Complexity Analysis:**

The space complexity of the algorithm is O(n^2) since we use a 2D boolean array of size (n+1) x (n+1) to store the intermediate results of the subproblems.

**Time Complexity Analysis:**

The algorithm has a time complexity of O(n^3), where n is the length of the input string. This is because the algorithm involves iterating over the length of the substring, and for each substring, there is an additional loop that checks all possible combinations of splitting the substring.

**Output:**

**((()))**

when you run this code, it will correctly return the balanced parenthesis string. For example, if you run the code with the argument n=3, it will return the string **"((()))".**