1. [10*points*] Say you have an $N \times N$ matrix A.

   a. [2.5 + 2.5] Please write the following two functions for it. Also describe their time complexity. A is globally/objectively accessible to the two functions:

   - $def setValue(x, y)$: *sets a value at row x and column y in A*
   - $def subRectangleSum(x1, y1, x2, y2)$: *calculates the sum value of rectangle encased within the two given point* $(x1, y1)$ *and* $(x2, y2)$. *Eg. Let*

   $$A = [\,[1,2,1,2] \\ [2,1,2,1] \\ [1,2,1,2] \\ [2,1,2,1]]$$

   Then $subRectandgleSum(1, 1, 2, 3)$ *returns* 9, *and* $subRectangleSum(0, 0, 2, 1)$ *returns* 9

   Write pseudocode. Explanation of your algorithm and clearly stating and explaining the complexity are mandatory

Algorithm 1 setValue

Sets the value of the element at row x and column y.
Input: $A[N][N]$ *an* $N \times N$ *matrix*
Input: $x, y < N$ *Integer coordinates*
Input: v *Integer value to set A to at position x, y*
Output: *None*
Time Complexity: $O(1)$
Space Complexity: $O(1)$

```
1: function SETVALUE($x, y, v, A$)
2:     $A[x][y] \leftarrow v$
3: end function
```

The above algorithm takes an (x, y) coordinate, a value v and an $N \times N$ matrix A. The x and y values are used to set the element at $A[x][y]$ to the value v. Because of direct memory access, since A is an $N \times N$ array, the operation is completed in constant time and space. Furthermore, as this function mutates A, nothing is returned.

Algorithm 2 subRectangleSum

Returns the sum of the elements in the sub-rectangle of A with top-left corner (x_1, y_1) and bottom-right (x_2, y_2).
Input: $A[N][N]$ *an* $N \times N$ *matrix*
Input: $x_1, x_2, y_1, y_2 < N$ *Integer coordinates*
Output: *sum Integer sum of elements*
Time Complexity: $O(n^2)$
Space Complexity: $O(1)$

```
1: function subRectangleSum($x_1, y_1, x_2, y_2, A$)
2:     $sum \leftarrow 0$
3:     for $i \leftarrow x_1$ to $x_2 + 1$ do                ▷ $O(n)$
4:         for $j \leftarrow y_1$ to $y_2 + 1$ do            ▷ $O(n^2)$
5:             $sum \leftarrow A[i][j]$
6:         end for
7:     end for
8:     return $sum$
9: end function
```

The above algorithm takes two coordinates in the form (x_1, y_1), (x_2, y_2) and an $N \times N$ matrix A then returns the sum of the values contained in the rectangle defined by the supplied coordinates.

In this rectangle, (x_1, y_1) defines the position of the top left corner and (x_2, y_2) defines the position of the bottom right corner.

Since $x_1, x_2, y_1, y_2 \leq N$ and there are two for loops, the total complexity is $O(n^2)$. Furthermore, as only the value of sum is modified at each iteration of the inner for loop and there are no other operations the total space complexity is $O(1)$ constant.

b. [5] Now very carefully consider this condition. The queries made upon A are in an indefinite stream, such that you have to execute the two functions repeatedly. If, for example, $SubRectangleSum()$ runs in $O(n^3)$ time, that time would be repeated for as long as you have your stream of query executing. Your goal is to bring the time complexity of the function to $O(1)$. You can use an extra matrix (also globally available) of the same size. You can also write helper functions to aid you in the program. Write the new algorithm for helper function and $SubRectangleSum()$. The conditions of writing pseudocode and explanation stand as always.

Algorithm 3 createSumMatrix

Returns a matrix of the same size as A, where each element is the sum of all elements in the sub-rectangle of A with top-left corner $(0, 0)$ and bottom-right corner (i, j).

Input: $A[N][N]$ *an* $N \times N$ *matrix*

Output: $S[N][N]$ *an* $N \times N$ *matrix where each element is the sum of the elements above and to the left*

Time Complexity: $O(n^2)$

Space Complexity: $O(n^2)$

```
function createSumMatrix($A$)
    $n \leftarrow A.length$
    $S[0 \dots N][0 \dots N] \leftarrow 0$
    $S[N][N] \leftarrow$ each element initialized to $0$
    $S[0][0] \leftarrow A[0][0]$
    for $i \leftarrow 1$ to $n$ do                                    ▷ $O(n)$
        $S[i][0] \leftarrow S[i-1][0] + A[i][0]$
        $S[0][i] \leftarrow S[0][i-1] + A[0][i]$
    end for
    for $i \leftarrow 1$ to $n$ do                                    ▷ $O(n)$
        for $j \leftarrow 1$ to $n$ do                                ▷ $O(n^2)$
            $S[i][j] \leftarrow S[i-1][j] + S[i][j-1] - S[i-1][j-1] + A[i][j]$
        end for
    end for
    return $S$
end function
```

createSumMatrix is the first helper function needed to solve the problem of a constant stream of queries being bottlenecked by the slow speed of our naive first attempt at *subRectangleSum*.

createSumMatrix first initializes an empty matrix S with the same size as A. S will contain a new matrix where each element is the sum of the elements above it or to the left of it. To achieve this we calculate the sum of the first row and column to be used in a cascading summation. As each element in S is the sum of the element which appears before it (row-wise or column-wise) and itself we create the first row and column so the second loop can consider the element previous $(i-1)$ without causing an out of bounds error.

We loop through the rest of A starting at $A[1][1]$ calculating each element of S as explained above. At each iteration we subtract $S[i-1][j-1]$ to make sure our cascading sum does not double count.

The first for loop has a complexity of $O(n)$ which is overshadowed by the second set of nested for loops which have a combined complexity of $O(n^2)$.Furthermore since $createSumMatrix$ initializes and then returns a new $N \times N$ matrix the total space complexity is $O(n^2)$.

At first glance this function may not seem to help our situation since the complexity is the same as our naive approach. This function will however only be called once. By creating this matrix one time we can then calculate the sum of any rectangle in A by taking $S[x_2][y_2]$ which is the sum of all of the elements in the rectangle from $(0,0)$ to (x_2, y_2), then subtracting the elements above and to the left of our rectangle. Note that since the area diagonally to the left of the sub-rectangle is contained in both subtractions it needs to be added back to maintain a correct sum. Once this is completed we can access any sum in $O(1)$ time as we will see below.

Algorithm 4 subRectangleSum

Returns the sum of the elements in the sub-rectangle of A with top-left corner (x_1, y_1) and bottom-right (x_2, y_2).
Input: *$S[N][N]$ an $N \times N$ matrix where each element is the sum of the elements above and to the left*
Input: *$x_1, x_2, y_1, y_2 < N$ Integer coordinates*
Output: *sum Integer*
Time Complexity: $O(1)$
Space Complexity: $O(1)$

```
function subRectangleSum($x1, y1, x2, y2, S$)
    if $x_1 = 0$ and $y_1 = 0$ then
        return $S[x_2][x_2]$
    else if $x_1 = 0$ then
        return $S[x_2][y_2] - S[x_2][y_1 - 1]$
    else if $y_1 = 0$ then
        return $S[x_2][y_2] - S[x_1 - 1][y_2]$
    else
        return $S[x_2][y_2] - S[x_1 - 1][y_2] - S[x_2][y_1 - 1] + S[x_1 - 1][y_1 - 1]$
    end if
end function
```

We see there are four cases to consider: when the rectangle comprises the entirety of A, when the rectangle is bounded on the left of A, when the rectangle is bounded at the top by A and when the rectangle is fully contained in A.

Since this function performs a simple sum of two to four elements it has a complexity of $O(1)$. To make this function possible we first had to create a summation matrix with the function *$createSumMatrix$* so while this function has a space complexity of $O(1)$ the combined complexity with *$createSumMatrix$* rises to $O(n^2)$.

2. [5*Point*] Please read through 15.2 from CLRS text book, which describes algorithm for Balanced Parenthesis in conjunction with Matrix Chain Multiplication. Please give a detailed algorithm to create Balanced Parenthesis using Dynamic Programming. Clearly state the time complexity and space complexity of the proposed algorithm.

To give an algorithm to create balanced parentheses for a multiplication chain of matrices we modify the algorithm from section 15.2 of CLRS which calculates the minimum number of multiplications needed to solve the chain.

Algorithm 5 MemoizeMatrixChain

Prints balanced parentheses for a matrix multiplication chain which represent the minimum number of operations needed to perform the multiplication.

Input: $p[n+1]$ An array of $N+1$ numbers representing the dimensions of the matrices in the multiplication chain.

Input: n The number of matrices in the chain.

Output: r The minimum number of operations needed to complete the multiplication.

Time Complexity: $O(n^3)$

Space Complexity: $O(n^2)$

```
1: function MemoizedMatrixChain($p, n$)
2:     $M[0 \dots n][0 \dots n] \leftarrow \infty$
3:     $P[0 \dots n][0 \dots n] \leftarrow 0$
4:     $r \leftarrow LookupChain(M, P, p, 1, n-1)$          $\triangleright O(n^3)$
5:     $PrintOptimalParens(P, 1, n-1)$                      $\triangleright O(n^2)$
6:     print($newline$)
7:     return $r$
8: end function
```

We add two things to the algorithm in CLRS to facilitate the creation of balanced parentheses: A new $n \times n$ matrix which will store the indices of the parentheses to be added to the chain to create the optimal ordering, and a call to our helper function *PrintOptimalParens*() which will print the multiplication chain with the required parentheses using the indices stored in P.

Algorithm 6 LookupChain

returns the minimum number of scalar multiplications needed to compute the matrix $A[i \dots j] = A[i]A[i+1] \dots A[j]$.
Input: $M[n]$ An $n \times n$ table such that $m[i][j]$ = minimum number of scalar multiplications needed to compute the matrix $A[i \dots j] = A[i]A[i+1] \dots A[j]$. **Input:** $P[n]$ An $n \times n$ table used to construct an optimal solution where $P[i][j]$ = index of the sub-problem from which we split the product $A[i \dots j]$ when computing the optimal ordering.
Input: $p[n+1]$ An array of $N+1$ numbers representing the dimensions of the matrices in the multiplication chain.
Input: $i, j < n$ The starting and ending indices.
Output: r The minimum number of operations needed to complete the multiplication.
Time Complexity: $O(n^3)$
Space Complexity: $O(n^2)$

```
function LookupChain($M, P, p, i, j$)
    if $M[i][j] < \infty$ then
        return $M[i][j]$
    else if $i = j$ then
        return 0
    else
        for $k \leftarrow i$ to $j$ do
            $q \leftarrow LookupChain(M, P, p, i, k)$
            $q \leftarrow q + LookupChain(M, P, p, k+1, j)$
            $q \leftarrow q + p[i-1] * p[k] * p[j]$
            if $q < M[i][j]$ then
                $M[i][j] \leftarrow q$
                $P[i][j] \leftarrow k$
            end if
        end for
        return $M[i][j]$
    end if
end function
```

Next we have the *LookupChain*() function described in CLRS. The only addition to this function we have made is to pass the P table (described above) to the function and to set the value of $P[i][j]$ to k when an optimal ordering is found.

As we have only added a single operation the time complexity stays $O(n^3)$ as described in the text.

Algorithm 7 PrintOptimalParens

Prints the optimal parenthesization of matrix chain product $A[i]A[i+1]\dots A[j]$

Input: $P[n]$ An $n \times n$ table used to construct an optimal solution where $P[i][j]$ = index of the sub-problem from which we split the product $A[i \dots j]$ when computing the optimal ordering.

Input: $i, j < n$ The starting and ending indices.

Output: *None*

Time Complexity: $O(n)$

Space Complexity: $O(1)$

```
function PrintOptimalParens($P, i, j$)
    if $i = j$ then
        print($char(64 + i)$, $end = ' '$)
    else
        print($'('$, $end = ' '$)
        $PrintOptimalParens(P, i, P[i][j])$
        $PrintOptimalParens(P, P[i][j] + 1, j)$
        print($')'$, $end = ' '$)
    end if
end function
```

PrintOptimalParens() recursively goes through the table P printing the parentheses based on their optimal position memoized in *LookupChain*(). As *PrintOptimalParens*() only considers the diagonal, at most the worst case complexity of the function will be $O(n)$

Since the complexity of *LookupChain*() is $O(n^3)$ and *PrintOptimalParens*() has a complexity of $O(n)$ the total complexity of *MemoizeMatrixChain*() is $O(n^3 + n)$ where the $O(n)$ term is insignificant in comparison to the $O(n^3)$ growth rate leaving the total complexity at $O(n^3)$