T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE4077 – Advanced Data Structures – Fall 2023

Project 1 - Report

Group Members

150119639 – Erdem PEHLİVANLAR

150121823 – Emre Gürkan

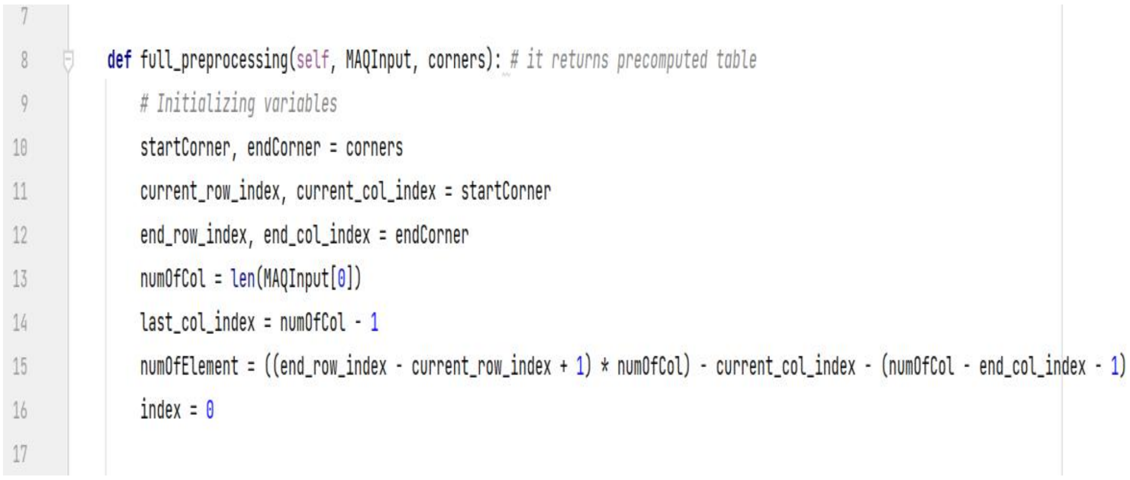
Full Preprocessing:

- Solution Approach

  Above all, Full Preprocessing is used in the offered solution to optimize MAQ queries. As part of the preprocessing, a precomputed table containing the minimum items for each conceivable subrectangle inside the input array is created. The design of this table makes efficient querying possible.

  - Initialization:

    The precomputed table, the start and end corners of the query areas, the input array, other required variables and data structures are all initialized. Morever, all elements in 2D array input are stored in 1D virtual array. Virtual array means that there is no 1D array, but there is an 1D array structure by using index calculations.



```python

    def full_preprocessing(self, MAQInput, corners): # it returns precomputed table
        # Initializing variables
        startCorner, endCorner = corners
        current_row_index, current_col_index = startCorner
        end_row_index, end_col_index = endCorner
        numOfCol = len(MAQInput[0])
        last_col_index = numOfCol - 1
        numOfElement = ((end_row_index - current_row_index + 1) * numOfCol) - current_col_index - (numOfCol - end_col_index - 1)
        index = 0

```

numOfElement: That is a formula which calculates number of elements for given region in 1D virtual array. It provides a dynamic structure for preprocessing.

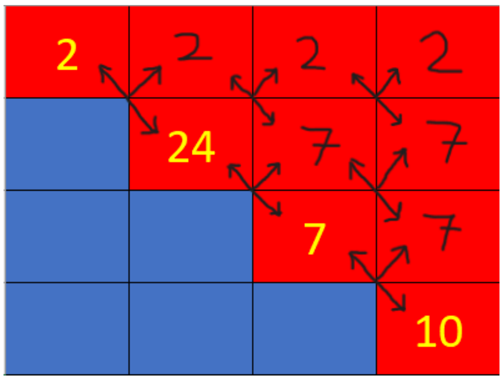
  - Precomputed Table Construction:

    The solution constructs a precomputed table by iterating through the input array and filling the table with array elements as cross firstly. Then, precomputing process is applied via current precomputed table recursively.

    Algorithm starts from first row and first column and it continues by comparing two lower cross cells with respect to minimum. Then, It assigns the result to upper cross cell up until it reachs the last column of the precomputed table like **Fig.1**.

Sample Table:

Sample Input: $\begin{bmatrix} 2 & 24 \\ 7 & 10 \end{bmatrix}$



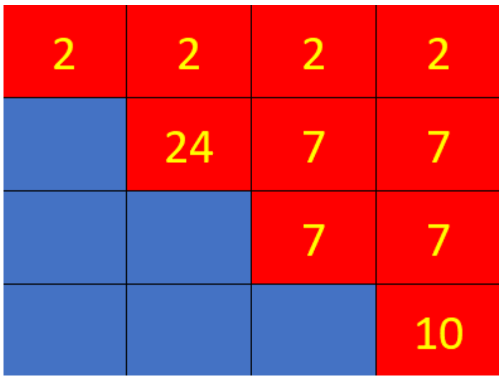


Fig.1 **Fig.2**

This process is done in a manner that allows for quick access during queries.

- Query Processing:

  MAQ queries are processed using the precomputed table. For a given query region, the solution directly looks up the minimum element in the precomputed table, providing a fast and accurate response to the query.

- Time and Space Complexities
  - Time Complexity

    Precomputed table is square matrix, so

    n = row number or column number of precomputed table

$$\sum_{i=1}^{n} i \;=\; n \times (n+1) \div 2 \;=\; O((n^2+n) \div 2) \;=\; O(n^2)$$

Time complexity of preprocessing process is $O(n^2)$.Time complexity of query process is O(1). In conclusion, $\langle O(n^2), O(1) \rangle$

- ### Space Complexity

  Algorithm stores only precomputed table.

  m = row number of input

  k = column number of input

  n = number of elements in 2D array input = m x k

  Precomputed table has n rows and n columns, so number of elements of precomputed table is, n x n = n^2. In conclusion, space complexity of that algorithm is O(n^2)

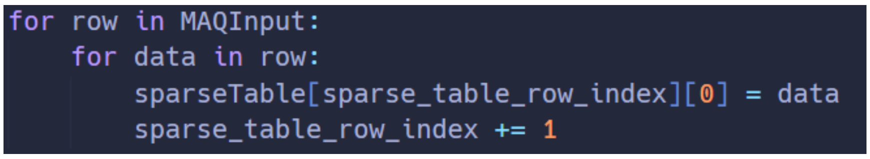
Sparse Table:

- ### Solution Approach

  Sparse Table algorithm for MAQ problem acts as if it was designed for RMQ problem without sacrificing space complexity.

  - #### Initialization:

    The algorithm starts by filling the first column of the sparse table with the flattened matrix input.



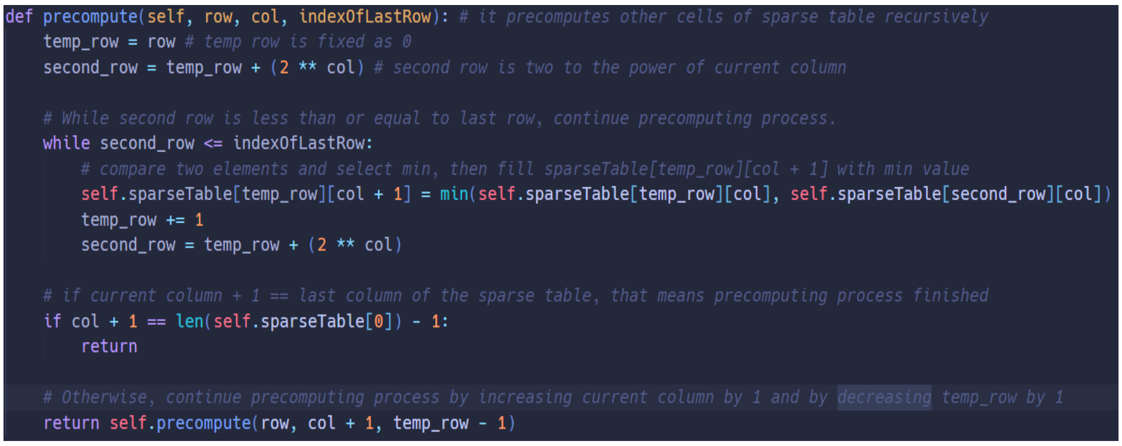
```
for row in MAQInput:
    for data in row:
        sparseTable[sparse_table_row_index][0] = data
        sparse_table_row_index += 1
```

An example:

Assume the matrix [[1,2],[3,4]], after the initialization, `sparseTable` would be [[1,2,3,4]].

  - #### Sparse Table Construction:

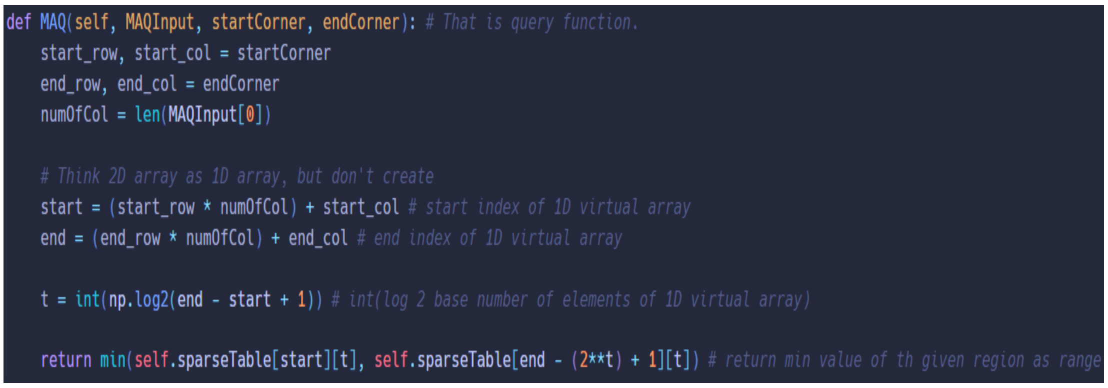
    This dynamic programming algorithm works recursively to generate the sparse table.



```python
def precompute(self, row, col, indexOfLastRow): # it precomputes other cells of sparse table recursively
    temp_row = row # temp row is fixed as 0
    second_row = temp_row + (2 ** col) # second row is two to the power of current column

    # While second row is less than or equal to last row, continue precomputing process.
    while second_row <= indexOfLastRow:
        # compare two elements and select min, then fill sparseTable[temp_row][col + 1] with min value
        self.sparseTable[temp_row][col + 1] = min(self.sparseTable[temp_row][col], self.sparseTable[second_row][col])
        temp_row += 1
        second_row = temp_row + (2 ** col)

    # if current column + 1 == last column of the sparse table, that means precomputing process finished
    if col + 1 == len(self.sparseTable[0]) - 1:
        return

    # Otherwise, continue precomputing process by increasing current column by 1 and by decreasing temp_row by 1
    return self.precompute(row, col + 1, temp_row - 1)
```

The `second_row` variable is for the difference of indexes that the algorithm will compare and find the minimum. For the first run it is 1 then it goes 2, 4, 8 and so forth accordingly. This algorithm is essentially the same as the RMQ problem algorithm.

- Query Processing:



```python
def MAQ(self, MAQInput, startCorner, endCorner): # That is query function.
    start_row, start_col = startCorner
    end_row, end_col = endCorner
    numOfCol = len(MAQInput[0])

    # Think 2D array as 1D array, but don't create
    start = (start_row * numOfCol) + start_col # start index of 1D virtual array
    end = (end_row * numOfCol) + end_col # end index of 1D virtual array

    t = int(np.log2(end - start + 1)) # int(log 2 base number of elements of 1D virtual array)

    return min(self.sparseTable[start][t], self.sparseTable[end - (2**t) + 1][t]) # return min value of th given region as range
```

After getting the wanted region, the algorithm calculates the start and the end indexes since our data structure was originally a matrix which it treats as a vector. After the calculation we query the sparse table in the same manner as it is done in the RMQ problem.

- Time and Space Complexities
  - Time Complexity

At the initialization step, the algorithm flattens the given matrix which has **θ(n^2)** time complexity.

Preprocessing is done in the same manner as it is in the RMQ problem so it does not change, **O(nlogn).**

Querying process has some tiny calculations which should not affect complexity in big-oh scales so it is, **O(1).** In conclusion, ⟨O(n^2), O(1) ⟩

- Space Complexity

  Even though we treat the matrix as it is a vector we do not allocate extra memory for it. So the space complexity remains, O(nlogn).

  Keep in mind that n is the total number of elements in the given matrix.

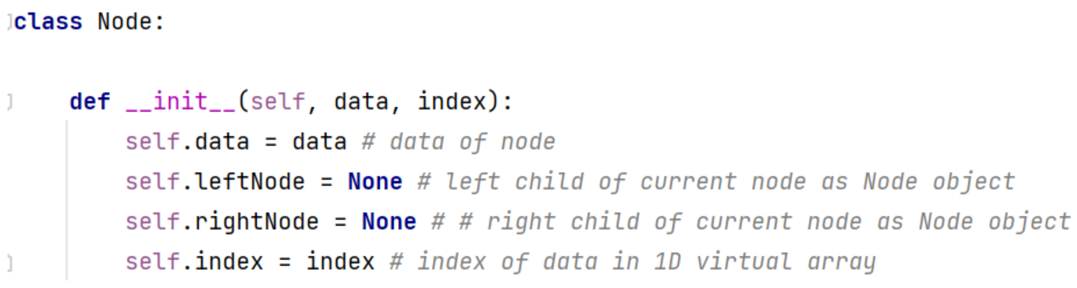
Cartesian Tree:

- Solution Approach

  The binary tree that results from a series of unique items is called a Cartesian tree. Its singular characteristic is that the input sequence's minimum element is always the tree's root.

  - Initialization:

    Each node in the tree is represented as an "Node" object. Node class has 4 attributes called as data, leftNode, rightNode and index. Data is represented input element as integer. LeftNode is represented left child of current node as Node object. RightNode is represented right child of current node as Node object same way and index indicates the index of element in 1D virtual input array.

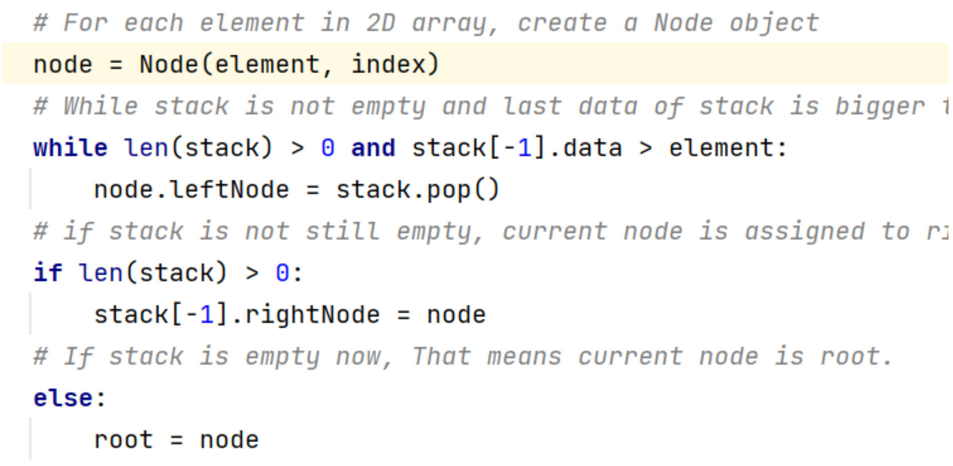


```
class Node:

    def __init__(self, data, index):
        self.data = data # data of node
        self.leftNode = None # left child of current node as Node object
        self.rightNode = None # # right child of current node as Node object
        self.index = index # index of data in 1D virtual array
```

  - Cartesian Tree Construction:

    The 2D input array's items are iterated through by the technique.

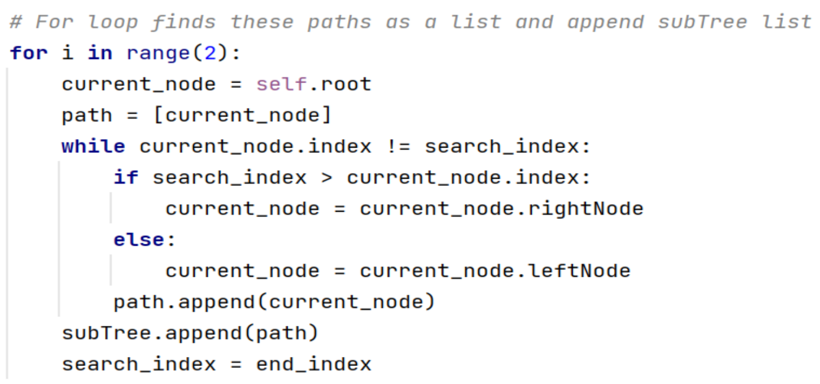
A Node object is produced for every element, containing the element's value and matching index in 1D virtual input array. Nodes are removed from the stack and designated as the left child of the current node as long as the stack is not empty and the current element is smaller than the final element in the stack. The current node becomes the right child of the final element in the stack if it isn't empty at this point. The current node is designated as the Cartesian Tree's root if the stack is empty. After that, the active node is added to the stack to begin processing further data.To keep the members of the Cartesian Tree in their proper positions, the index variable is increased.The method returns the root of the created Cartesian tree after processing each element.



```
# For each element in 2D array, create a Node object
node = Node(element, index)
# While stack is not empty and last data of stack is bigger t
while len(stack) > 0 and stack[-1].data > element:
    node.leftNode = stack.pop()
# if stack is not still empty, current node is assigned to ri
if len(stack) > 0:
    stack[-1].rightNode = node
# If stack is empty now, That means current node is root.
else:
    root = node
```

- [Query Processing:]()

In the query processing, there is a method called as searchTree(start_index, end_index). This method finds the paths that reach two nodes which have start and end index from root. Start and end index represents the query range. Then, it finds the intersection node of these two nodes. Intersection node represents the minimum element of given range in the 2D input array.



```
# For loop finds these paths as a list and append subTree list
for i in range(2):
    current_node = self.root
    path = [current_node]
    while current_node.index != search_index:
        if search_index > current_node.index:
            current_node = current_node.rightNode
        else:
            current_node = current_node.leftNode
        path.append(current_node)
    subTree.append(path)
    search_index = end_index
```

Sample Query:

Sampe input:

$$\begin{matrix} 51 & 24 & 3 \\ 68 & 41 & 54 \\ 11 & 34 & 53 \end{matrix}$$

1D **virtual** array: [51, 24, 3, 68, 41, 54, 11, 34, 53]

If query is MAQ((1, 0), (2, 2)), start_index is 3 and end_index is 8 in 1D **virtual** array.

Algorithm finds two paths that reach start and end node from the root like **Fig.3** and **Fig.4**.

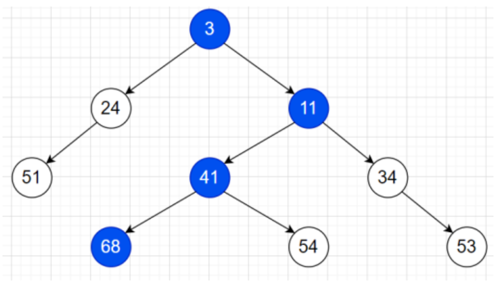


Fig.3

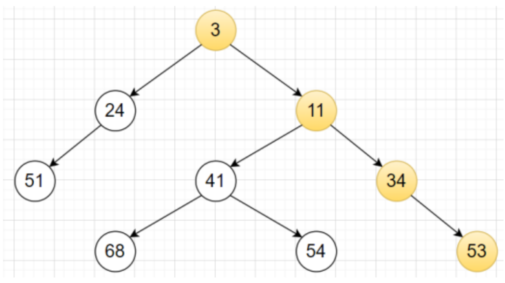


Fig.4

As it seems in the **Fig.5**, intersection node of these two paths is **11** that is **red labeled**. That node is the minimum element of the given query area.

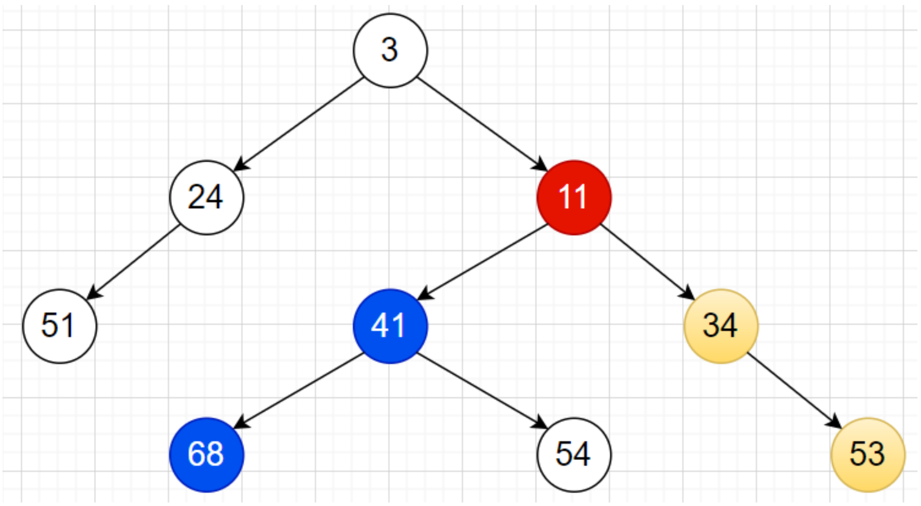


Fig.5

- Time and Space Complexities
    - Time Complexity

      n = number of elements in 2D array

      Building the Cartesian tree by iterating over each element in the 2D array:

      The stack size, which is constrained by the amount of elements, determines how the while loop inside the loop operates.When building the tree, the loop's time complexity is still O(n).Thus, O(n) is the temporal complexity of constructCartesianTree().

      Creating pathways to the start and end nodes: The while loop inside the for loop traverses from the root to the leaf node, going as far as the tree allows it to go, which in the worst case scenario may be log(n). Both pathways are constructed in O(log(n)) time.

      Locating the point where the paths intersect:

      Up to the shorter path's length, the for loop continues. Its temporal complexity is O(log(n)) in the worst scenario.Consequently, searchTree() has an O(log(n)) time complexity.

      Time complexity of query process is O(log(n)) because of that search tree is used in query process.

      In conclusion, ⟨O(n), O(log(n))⟩

    - Space Complexity

      n = number of elements in 2D array

      The stack, which at most will contain the number of elements in the input 2D array, is the primary factor determining the space complexity of the Cartesian Tree generation process. Consequently, the stack's space complexity is O(n).