Data Structures and Algorithms, Summative Assignment 1.

Conditions.

You are given an n x m matrix with integer entries that has the following properties: (1) Each row has a unique maximum value, (2) If the maximum value in row i of the matrix is located at column j, then the maximum value in row i+1 of the matrix is located at a column k, where k ¿ j.

Question 1.

Question 1. Write a function maxIndex that finds the index of the maximum entry of a row between columns with indices start and end inclusively. The row is given as an array named 'row'.

```
public static int maxIndex(int[] row, int start, int end)
{
    int index = start;
    for (int i = start; i <= end; i++)
    {
        if (row[i] > row[index])
        {
            index = i;
        }
    }
    return index;
}
```

My solution shown above has the average and worst case time complexity of O(n) because it employs a linear search. The for loop compares each subsequent value in the array with the first value in the array at index 0 until it finds a larger value, which in the average and worst-case the for loop would loop n times comparing all values with the current maximum; therefore, it is in O(n) time complexity.

Question 2.

Question 2. A rectangular block of a matrix is given by a row and column of the upper-left corner in startRow and startCol, and row and column of the lower-right corner endRow and endCol, such that startRow ¡= endRow and startCol ¡= endCol. Write a blockMaxValue function that finds the value of the maximum entry of a given block assuming that the block satisfies the properties (1) and (2) above.

```java
public static int blockMaxValue(int[][] matrix, int startRow, int startCol, int endRow, int endCol)
{
    if (startRow > endRow || startCol > endCol)
    {
        return Integer.MIN_VALUE;
    }

    int midRow = (startRow + endRow) / 2;
    int maxColIdx = maxIndex(matrix[midRow], startCol, endCol);
    int value = matrix[midRow][maxColIdx];

    int upperMax = blockMaxValue(matrix, startRow, startCol, midRow - 1, maxColIdx);
    int lowerMax = blockMaxValue(matrix, midRow + 1, maxColIdx, endRow, endCol);

    return Math.max(value, Math.max(upperMax, lowerMax));
}
```

Question 3.

Question 3. Write a function matrixMaxValue that finds the maximum value of a matrix that satisfies properties (1) and (2) above and provides a better worst-case time complexity than O(nm).

```java
public static int matrixMaxValue(int[][] matrix)
{
    int endRow = matrix.length;
    int endCol = matrix[0].length;

    return blockMaxValue(matrix, 0, 0, endRow - 1, endCol - 1);
}
```

My solution shown above calls blockMaxValue with the parameters of 0,0 (start) to endRow-1,endCol-1 (end) so it has the same time complexity as blockMaxValue which has the average and worst case time complexity of O(n.log(m)) using a divide and conquer approach.

- The line: If `startRow > endRow` or `startCol > endCol` takes a constant time, $O(1)$.
- If the matrix size is 1×1, the function returns a single element. This also takes a constant time, $O(1)$.
- The `maxIndex` function finds the maximum value in the middle row within the current column range (`startCol` to `endCol`).
- This function iterates through the columns in the middle row, which takes $O(m)$ time, where m is the number of columns.
- The `blockMaxValue` function makes two recursive calls:
  - `Upper Submatrix:` From `startRow` to `midRow - 1` and `startCol` to `maxColIdx`.
  - `Lower Submatrix:` From `midRow + 1` to `endRow` and `maxColIdx` to `endCol`.
- Each call reduces the problem size by half. So, the depth of recursion is $O(\log n)$, where n is the number of rows.

Total Time Complexity Calculation

To calculate total time complexity, we need to combine the complexities of each step: `Finding the Maximum in the Middle Row:`

- This step takes $O(m)$ time.

`Recursive Calls:`

- The number of recursive calls is two, and each call deals with a smaller submatrix of half the size in terms of rows.
- The depth of recursion is $O(\log n)$.

The total time complexity can be represented as the following function:

$$F(n, m) = m + 2F\left(\frac{n}{2}, m\right)$$

The $2F\left(\frac{n}{2}, m\right)$ term represents the time taken by the two recursive calls. Since the recursion depth is $O(\log n)$: Our next step is to expand and substitute to calculate the total time complexity:

$$F(n, m) = 2F\left(\frac{n}{2}, m\right) + O(m)$$

$$F\left(\frac{n}{2}, m\right) = 2F(n/4, m) + O(m)$$

$$F(n, m) = 2[2F\left(\frac{n}{4}, m\right) + O(m)] + O(m)$$

$$4F\left(\frac{n}{4}, m\right) + 2O(m) + O(m)$$

$$4F\left(\frac{n}{4}, m\right) + 3O(m)$$

Finding F(n/4, m):

$$F\left(\frac{n}{4}, m\right) = 2F\left(\frac{n}{8}, m\right) + O(m)$$

$$F(n, m) = 4[2F\left(\frac{n}{8}, m\right) + O(m)] + 3O(m)$$

$$= 8F\left(\frac{n}{8}, m\right) + 4O(m) + 3O(m)$$

$$= 8F\left(\frac{n}{8}, m\right) + 7O(m)$$

From the calculations we observe a pattern of:

$$F(n, m) = 2^k F(n/2^k, m) + (2^k - 1)O(m)$$

Since at each recursion n halves, it reaches the base case $T(1, m)$ when $n/2^k = 1$ where $2^k = n$, so $k = log(n)$ Substituting

$$k = log(n)$$

$$F(n, m) = 2^log(n)F(1, m) + (2^log(n) - 1)O(m)$$

Since $2^log(n) = n$ and assuming $F(1, m)$ is a constant:

$$F(n, m) = nO(1) + (n - 1)O(m)$$

$$= O(n) + O(nm - m)$$

$$= O(nm) - O(m)$$

Since $O(m)$ is negligible compared to $O(nm)$ we simplify to:

$$F(n, m) = O(m \log n)$$