Sreeram Danda
CS577 Homework #2
Problem #2

a)

```
boolean search(int[][] matrix, int target){
	int middleRow = Math.floor(matrix.length/2);
	int middleCol = Math.floor(matrix[0].length/2);
	int midVal = matrix[middleRow][middleCol];

	//If middle element is target return true
	if(midVal == target){
		return true;
	} else {
		//Always check top right
		boolean isTopRight =
			search(matrix[0...middleRow][middleCol...m], target);
		if(isTopRight) return true;

		//If not top right and target is bigger, check bottom half
		if(!isTopRight && target > midVal){
			return search(matrix[middeRow...n][0...m], target);
		//Otherwise target is smaller, check left half of matrix
		} else if(!isTopRight && target < midVal){
			return search(matrix[0...n][0...middleCol], target);
		}
	}
}
```

Program Correctness

Base Case: The middle element is the target, the program correctly returns true.

Recursive Case:

If the middle element is not the target. The program must check the top right quarter of the matrix. This is because the elements of the matrix increase row-wise and col-wise. Therefore, if the target is greater than the middle element, there is no guarantee that the element is less than all elements in the rows prior to the middle element. Similarly, if the target is less than the middle element, there is no guarantee that the middle element is greater than all the elements to the right of it in the columns above it. The algorithm then makes a recursive call on the matrix with a quarter of the size. Thus, by induction we can assume the recursive call on the top right quarter of the matrix returns correctly. If the target element is not in the top right, then if the target is greater than the middle element, the algorithm makes a recursive call on the bottom half of the matrix. Since the target is greater than the middle element, it is also greater than all elements matrix[i][j] where i < middleElement.row, and j < middleElement.col. Additionally, the algorithm already checks the top right prior to this call. Then again by the IH, the algorithm correctly returns for the recursive call on the bottom half of the array. Similarly, if the target element is less than the middle element the algorithm

makes a recursive call on the left half of the matrix. Since the algorithm has already checked the top right of the matrix. The only possible locations of the target exist on the left side of the matrix as all elements to the right are strictly greater than the middle element given the top right has been checked. Thus, by the IH the algorithm correctly returns for the recursive call on a matrix half the size.

b) To understand the number of comparisons in a decision tree model, we can begin the tree with the very first element from the matrix. This element is the smallest element in the matrix based on its structure. Then we make a comparison between the two adjacent elements and pick the smaller one. We continue to build the tree for each possible comparison. Note that the more elements that are picked the number of comparisons required to find the next smallest element continually increases. For example after finding the second smallest element there are three possible elements that can be the next smallest. This would require three comparisons. At the very end of all the comparisons we need to return the series of elements which are correctly sorted, which contains n^2 elements. Inorder to reach the bottom of the decision tree it takes, in the best case scenario, nlogn comparisons. Since there are n^2 elements it takes, $\Theta(n2 \log n)$. This means that it is not possible to sort the array in O(n) comparisons because in the worst case, it takes $\Theta(n2 \log n)$ comparisons to get to the bottom of the decision tree. Only making n comparisons would not allow us to obtain a completely sorted array as after n elements, the sorting of the n^2- n elements is completely arbitrary and not guaranteed to be sorted.

c) As noted in part b) even in the worst case the algorithm takes $O(n^2\log(n))$ time. In fact, because of the structure of the matrix it is impossible to create a sorted array without making comparisons for each element. For each element there are an arbitrary number of other elements that give no information without performing an operation, except for the elements in the same row or column. Therefore we cannot label this sorting as $o(n^2\log(n))$.