CSE 321 – Homework #4 Report

Yasir Nacak – 161044042

Q1.

a) On the basis:
a0,0 + a1,1 <= a0,1 + a1,0 -> Special(A2,2)

We need to show that:
For Every 1 <= i < m, 1 <= j < n(ai,j + ai+1,j+1 <= ai,j+1 + ai+1,j) -> Special (AM,N)

Induction Step:
For every 1 <= i < m, 1 <= j < n(ai,j + ai+1,j+1 <= ai,j+1 + ai+1, j)
For every j < n(am,j + am+1,j+1 <= am,j+1 + am+1,j) AND Special (AM,N)
➔ Special (AM+1,N), Similarly for A(M,N+1)

b) I firstly looked for the leftmost minimum elements for each row. For Special arrays, a rule is that the leftmost elements should always go in the right direction or stay where they are. This means that, if the 0^{th} element of the first row is its leftmost minimum, then we need to have an index greater or equal to 0 for the next row. An example of a correct sequence of leftmost minimum elements is:

[1, 2, 2, 3, 5, 5, 7]

In this example, we can see that the sequence of indices of leftmost minimums of each row is a non-decreasing sequence. If we have given a one-off Special array, we can simply look at the sequence of leftmost minimum indices. An example of a faulty by one element Special array may have the following leftmost minimum indices:

[1, 2, 1, 3, 5, 5, 6]

In this sequence, 2^{nd} index, or the 1 in the 3^{rd} element is faulty (shown by red). To fix this, we can simply make the second row's leftmost minimum element as its 2^{nd} or 3^{rd} elements. And we can do that by giving its 2^{nd} or 3^{rd} element a value lesser than the current minimum of that row.

Pseudocode:

```
procedure fix_non_special(array[rows, cols]):

        leftmost_mins = []

        for each row in array:
                leftmost_mins.add(min(row))
        end for

        faulty_row = -1
        faulty_element = -1

        foreach lm_min in leftmost_mins:
                if lm_min is smaller than its previous element
                        faulty_row = lm_min
                        faulty_element = leftmost_mins[lm_min]
                if lm_min is greater than its next element
                        faulty_row = lm_min
                        faulty_element = leftmost_mins[lm_min]
        end for

        new_min_value = array[faulty_row, faulty_element]
        new_min_index = leftmost_mins[faulty_row - 1]

        return faulty_row, new_min_index, new_min_value

end procedure
```

c) For this part, I designed an algorithm that splits the matrix by odd rows and even rows, then finds the leftmost minimum elements in them. But the split part is being done continuously up until we are left with one row, then the combining step that is required in divide-and-conquer algorithms begin. I also noticed the rule of non-decreasing leftmost minimum index rule in the Special Arrays so I designed my algorithm so that the next row starts to look for its minimum from the point that the previous row has its minimum.

d) For each odd row i, start at column j = f(i-1), where f(0) = 0; let mi = inf; iterate to j = f(i+1), where f(n+1) = m; setting mi to min(mi, ai,j).

Running time is:

$O(m)+\text{Sum}(i=1 \text{ to } m)\ f(i-1) – f(i+1) = O(m) + f(0) + f(m) + \text{Sum}(i=1 \text{ to } m)\ f(i)-f(i)$

$= O(m)+O(n)+0$

$=O(m+n)$

Q2.

For this question, I designed a divide-and-conquer algorithm that works with the following cases:

Let's have arrays X, Y and the index we want to find, K.

1. If center index of X + center index of Y < K
   a. If center element of X > center element of Y, discard first half of Y, adjust K
   b. Else, discard first half of X, adjust K
2. Else If center index of X + center index of Y > K
   a. If center element of X > center element of Y, discard the second half of X
   b. Else, discard the second half of Y

The running time of this algorithm can be analyzed such that, every time we do a recursive call, we reduce one of the array's size by the half of it. This means that we are doing log(length(array)) operations for each array until we reduced its size such that we have only the remaining element. If we apply this algorithm for merging of two arrays, we need to do this operation for the both. Which gives us: O(log(length(X) + length(Y))) worst-case running time.

Q3.

For this question, to design a divide and conquer algorithm, I decided to split the array into two parts and apply the solution to each of them. This way, we have the solution either in the left subarray or the right subarray. But there is also another case where we have the solution in the crossing (where middle element is in and some parts of left and right subarrays are also in). We can solve the only-left or only-right subproblems by simply calling our initial solution method but we need to come up with another solution for the crossing. To do that, I created a new function where I include the middle element, go until the leftmost index of the array, take all the numbers and update the sum and whenever we get an increase in the total sum, update the left sum. I also did the same for the right part where I started from the next element of the middle element, went up to the rightmost element, added all the elements into a sum and whenever I get a value greater than the current sum, I update the right sum. Applying this crossing sum gave me the value for the maximum contiguous subarray sum for the mix of left and right parts. After getting only-left, only-right and crossing sums, I find the maximum of them and went on.

The running time of this algorithm can be analyzed in the way that, for the left and right sums, we divide the array into two parts until we get only one element. This division can be done log(length(array)) times. For the crossing part, our algorithm goes through the whole array, so this gives us length(array) times of work. In total, if we call length(array) = n, the complexity is O (n log(n)) in the worst-case.

Q4.

I this part, I used graph coloring to find out if a graph is bipartite or not. A bipartite graph is only possible if we can color it in a way that there are not any neighbors with the same color. We are going to use two colors in our coloring process. I called those colors -1 and +1 and called 0 as no-color. Then my algorithm works as:

1. Assign color 1 to current vertex
2. Assign color -1 all neighbors of current vertex
3. For each neighbor of the current vertex, assign color 1 to its neighbors.
4. Continue doing this until all vertices are colored.
5. If in any stage, we want to color an already colored vertex as its opposite color, return False

The worst-case running time of this algorithm is the same as Breadth First Search since it utilizes the same traversal mechanism. The coloring step does not introduce any more complexity to the problem. In our case, the complexity of the BFS is since we are using an adjacency matrix, is O(V^2).

Q5.

For this part, I subtracted all the prices from the costs and get a profits array. Then in this array, I looked for the maximum value using divide and conquer.

Costs = [5, 11, 2, 21, 5, 7, 8, 12, 13, -]
Prices = [-, 7, 9, 5, 21, 7, 13, 10, 14, 20]
Profits = [2, -2, 3, 0, 2, 6, 2, 2, 7]

The recurrence relation used to find the maximum in the array is:

$T(n) = 2T(n / 2) + 1$

Which has the worst-case complexity O(n). Since iterating through costs and prices arrays also gives us O(n) time, the total worst-case time complexity is O(n + n) = O(n)