CSE 321 HW4 REPORT

1.

a.) Array is an $m\ x\ n$ array and we asked to prove that equation:

$$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$$

where i and j such that $1 \leq i < n$ and $1 \leq j < m$. If these are hold A is a special array. To prove that let's suppose

$$A[s,t] + A[s+1,t+1] \leq A[s,t+1] + A[s+1,t]$$

for all s and t such that $1 \leq s < n$ and $1 \leq t < m$ and also from definition for i, j, k, l $1 \leq i < k \leq n$ and $1 \leq j < l \leq m$.

If we consider $1 \leq t < m$ to prove that for rows:

$$\sum_{s=i}^{k-1} (A[s,t] + A[s+1,t+1]) \leq \sum_{s=i}^{k-1} (A[s,t+1] + A[s+1,t])$$

We can obtain these terms from this equation:

$$\begin{pmatrix} A[i,t] + A[i+1,t+1] \\ +A[i+1,t] + A[i+2,t+1] \\ +A[i+2,t] + A[i+3,t+1] \\ . \\ . \\ . \\ +A[k-1,t] + A[k,t+1] \end{pmatrix} \leq \begin{pmatrix} A[i,t+1] + A[i+1,t] \\ +A[i+1,t+1] + A[i+2,t] \\ +A[i+2,t+1] + A[i+3,t] \\ . \\ . \\ . \\ +A[k-1,t+1] + A[k,t] \end{pmatrix}$$

and if we cancel terms from both sides we end up with following equation:

$$A[i,t] + A[k,t+1] \leq A[i,t+1] + A[k,t] \qquad (1)$$

and for columns we can use same technique using equation (1) and we can totally prove the given statement.

This time we consider $1 \leq s < n$ to prove that for columns:

$$\sum_{t=j}^{l-1} (A[i,t] + A[k,t+1]) \leq \sum_{t=j}^{l-1} (A[i,t+1] + A[k,t])$$

We can obtain these terms from equation:

$$\begin{pmatrix} A[i,j]+A[k,j+1] \\ +A[i,j+1]+A[k,j+2] \\ +A[i,j+2]+A[k,j+3] \\ . \\ . \\ . \\ +A[i,l-1]+A[k,l] \end{pmatrix} \leq \begin{pmatrix} A[i,j+1]+A[k,j] \\ +A[i,j+2]+A[k,j+1] \\ +A[i,j+3]+A[k,j+2] \\ . \\ . \\ . \\ +A[i,l]+A[k,l-1] \end{pmatrix}$$

And from there we end up with following equation:

$$A[i,j]+A[k,l] \leq A[i,l]+A[k,j]$$

This the definition of Special array and proved.

b.) From the given equation we can find is there any contradiction against the special array rules by checking all pairs.

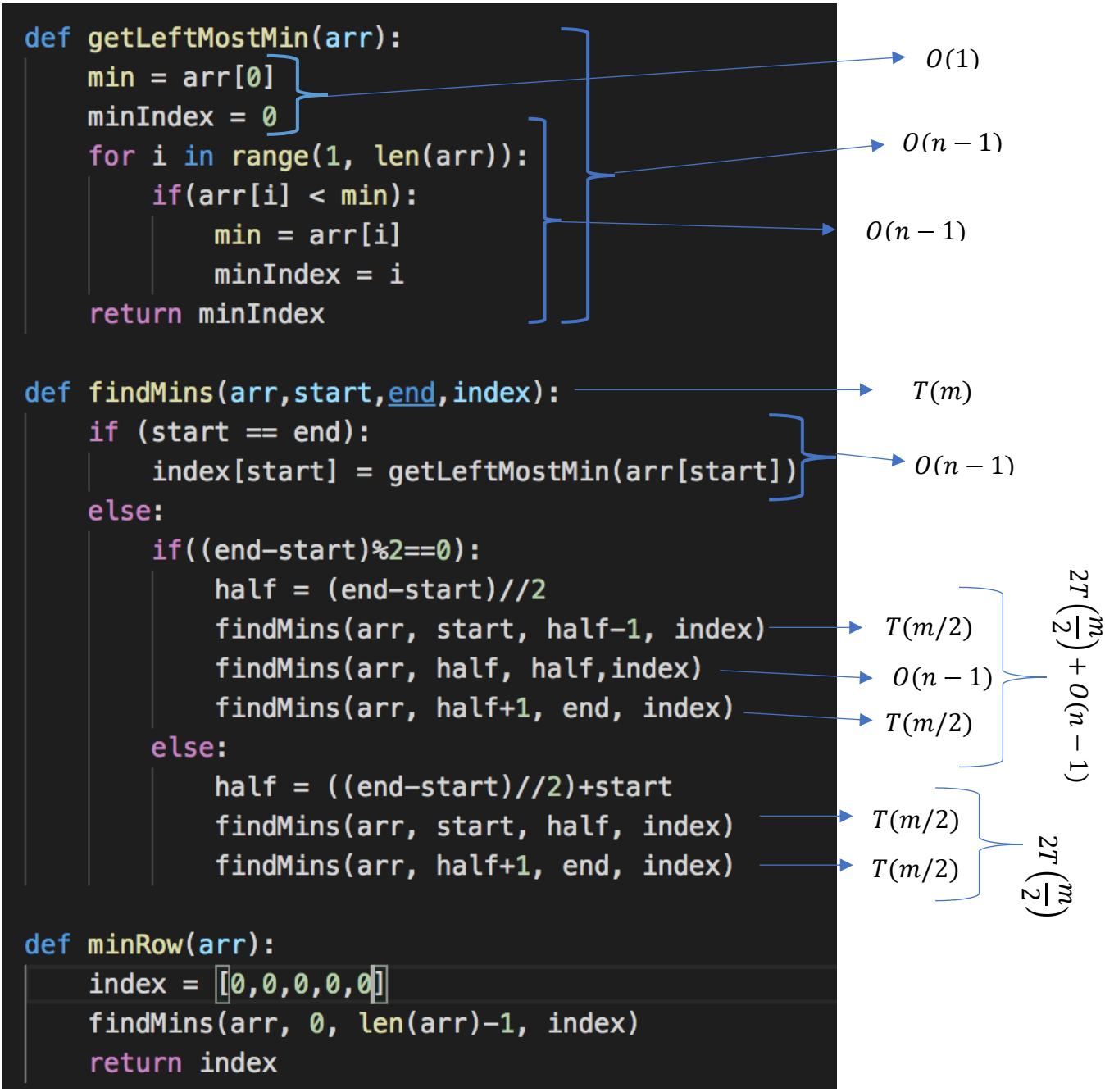
```
ALGORITHM makeSpecial(array[0…m][0….n])
    count→0
    for i →0 from i to m-1
            for j→0 from j to n-1
                    if (array[i][j]+array[i+1][j+1] > array[i][j+1]+array[i+1][j])
                            count→count+1
                            left→ array[i][j]+array[i+1][j+1]
                            right→ array[i][j+1]+array[i+1][j]
                            difference→left-right
                            array[i][j+1]→ array[i][j+1] + difference
    if (count == 1)
            return 1
    else
            return 0
```

This algorithm check every pair of elements for both side of equation and if we encounter such that array[i][j]+array[i+1][j+1] > array[i][j+1]+array[i+1][j], we must count that as a change and add the difference between sides to the first element of the right side. End of this algorithm, if we do only one change this means this is a successful for what asked and return otherwise return 0.

In python

```
def specialCheck(arr):
    count = 0
    for i in range(len(arr)-1):
        for j in range(len(arr[0])-1):
            if(arr[i][j]+arr[i+1][j+1] > arr[i][j+1]+arr[i+1][j]):
                count = count + 1
                left = arr[i][j]+arr[i+1][j+1]
                right = arr[i][j+1]+arr[i+1][j]
                dif = left - right
                arr[i][j+1] = arr[i][j+1]+dif
    if(count == 1):
        print2d(arr)
    else:
        print("It requires more than 1 change
```

c.) In this algorithm, I think if we find leftmost minimum element on each row individually, we reach the solution for whole array. So this is the base case, if special array only has one row. At this case we call getLeftMostMin function to gather the index of left most minimum element and we can push it into indexes array easily because we have the current row's index since when we computing that minimum element we act like there is only one row and row number is start or end. In other cases, we need to look is there even or odd number of rows in special array. If we have even number of rows we can easily divide array into two halves but if we don't have the even number of rows, this mean we need to extract middle row. When we extract middle row and send it to another recursive step, this will be the base case and will be computed easily. Since we extract middle element we can again divide list into two halves again and we can do recursive steps as in even number of rows. At the end for every row, we will reach the base case and we compute left most minimum element.



d.) If we consider m x n array for special array:
For the algorithm we have following recurrence relation:

$$T(m,n) = 2T\left(\frac{m}{2}\right) + O(n-1)$$

2. For that algorithm we use middle values of array to reach the solution. In if statements we check whether the one of the arrays is empty. If this is the case the remaining array's kth element will be the our searched element. This if statements not only for checking also when the recursive ends for one array it returns the kth array. This means process of finding kth element will continue until one of the array is empty. Also there is one if statement checks the k is equal to 0 because if the k is 0 this means we can look current arrays first start indexes and the smallest one will give us the kth element. I stored start and end index of array to avoid of overhead of slicing array. After base cases, we compute the middle indexes and elements for both arrays. If the k is greater than sum of middle indexes of two arrays this means kth element cannot be in one of the arrays left half. We can think that as if we merge two arrays smallest one will be come before our kth element. With this in mind there is no need to search kth element on the left half of smallest middle element. We divide one of the arrays size into 2. Since we decrease size of total arrays, we need to narrow searched kth element and decrease k by size of the decreasing size. The same logic for other half can be apply. If k less or equal than sum of middle indexes, greatest one of the middle elements will be come after kth element. So there is no need to search right half of that array. With that we can find kth element.

```
def findKth(arr1, start1, end1, arr2, start2, end2, k):
    if end1 - start1 < 0:
        return arr2[start2 + k]
    if end2 - start2 < 0:
        return arr1[start1 + k]
    if k < 1:
        return min(arr1[k + start1], arr2[k + start2])
    arr1MidIndex = (start1 + end1) // 2
    arr2MidIndex = (start2 + end2) // 2
    arr1Mid = arr1[arr1MidIndex]
    arr2Mid = arr2[arr2MidIndex]

    if ((arr1MidIndex - start1) + (arr2MidIndex - start2) < k):
        if(arr1Mid > arr2Mid):
            return findKth(arr1, start1, end1, arr2, arr2MidIndex+1, end2, k - (arr2MidIndex-start2) -1)
        else:
            return findKth(arr1, arr1MidIndex+1, end1, arr2, start2, end2, k - (arr1MidIndex-start1) - 1)
    else:
        if(arr1Mid > arr2Mid):
            return findKth(arr1, start1, arr1MidIndex-1, arr2, start2, end2, k)
        else:
            return findKth(arr1, start1, end2, arr2, start2, arr2MidIndex-1, k)

print (findKth(arr1,0,len(arr1)-1, arr2, 0, len(arr2)-1,2))
```

We can derive the following recurrence from code:

$$T(n,m) = \max\left(T\left(\frac{n}{2},m\right),T\left(n,\frac{m}{2}\right)\right) + 1$$

And this be equal to the

$$T(n,m) = \log(n) + \log(m)$$

Since if one of them fixed this will be log of other and since one of them is constant other one will show the complexity. We can also write it like

$$T(n,m) = max(\log(n) + \log(m))$$

3. In divide and conquer approach we divide total size into two. Let's say we want to find max subarray in Array[low……high]. This means we find middle point and then we recursively find maximum contiguous subarray for Array[low……..mid] and Array[mid+1……high]. This contiguous subarray Array[i…..j] can be in following intervals:

Array[low……mid],	$low \leq i \leq j \leq mid$
Array[mid+1……high],	$mid < i \leq j \leq high$
and can be cross on midpoint.	$low \leq i \leq mid \leq j \leq high$

We can find Array[low…..mid] and Array[mid+1….high] recursively because they are smaller size problems of real problem. For the crossing middle point. This is not a recursive step, it finds sums for left and right half and combines them as a single array. After all, between these three intervals bigger one will be the answer.

```
def maxCrossingSub(arr, low, mid, high):
    leftSum = float('-inf')
    total = 0
    maxLeft = 0
    maxRight = 0
    for i in range(mid, low, -1):
        total = total + arr[i]
        if (total > leftSum):
            leftSum = total
            maxLeft = i
    rightSum = float('-inf')
    total = 0
    for j in range (mid+1, high, 1):
        total = total + arr[j]
        if (total > rightSum):
            rightSum = total
            maxRight = j
    return maxLeft,maxRight,leftSum+rightSum

def findMaxSub(arr,low,high):
    if (high==low):
        return (low,high,arr[low])
    else:
        mid = (low+high)//2
        leftLow,leftHigh,leftSum = findMaxSub(arr,low,mid)
        rightLow,rightHigh,rightSum = findMaxSub(arr,mid+1,high)
        crossLow,crossHigh,crossSum = maxCrossingSub(arr, low, mid,high)
        if(leftSum >= rightSum and leftSum >= crossSum):
            return (leftLow,leftHigh,leftSum)
        if(rightSum >= leftSum and rightSum >= crossSum):
            return (rightLow,rightHigh,rightSum)
        else:
            return (crossLow,crossHigh,crossSum)
```

$\frac{n}{2}$

$\frac{n}{2}$

$\theta(n)$

$T(n)$

$T(\frac{n}{2})$

$T(\frac{n}{2})$

$\theta(n)$

The first function maxCrossingSub finds crossing array's sum and indices. We find sum for left half of middle and right half of middle. Then we combine them and return as start and end indices with sum of tow halves. This function take $\theta(n)$ times in total for both half of the entire array. findMaxSub function is the function where recursive steps occur. First it checks for the base case, that is there is only one element in array. Otherwise it firstly finds middle element position, after that as we explain above it find max sum for left and right halves. Also finds crossing midpoint if the max sum array intersect on middle element. Then function decides, Which one of the halves or crossing point bigger than others. And returns it sum and indices from start to end.

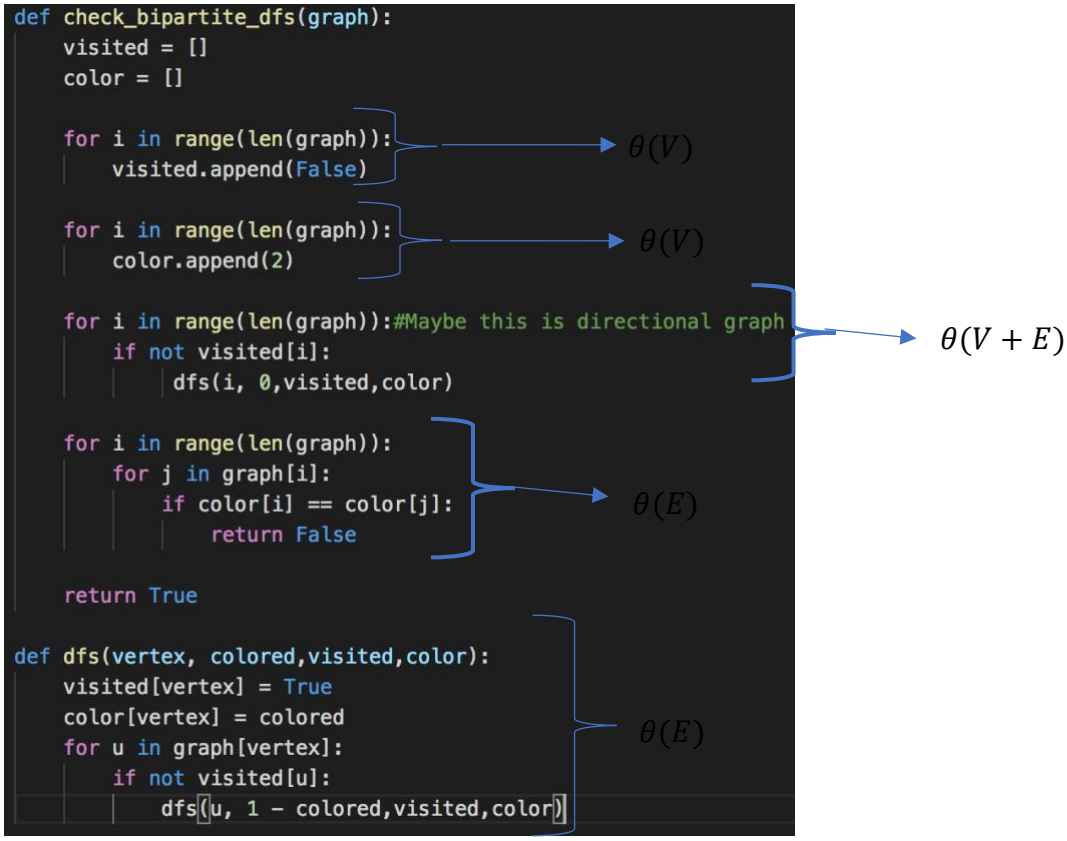
In base case algorithm takes $T(1) = \theta(1)$ times. After that if running time of algorithm is $T(n)$ we will have the following recurrence relation:

$$T(n) = \begin{cases} \theta(1) & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + \theta(n) & if\ n > 1 \end{cases}$$

We divide problem size to the 2 and this makes $T\left(\frac{n}{2}\right)$ also since we have now 2 problem we multiply it with 2. Crossing point function takes $\theta(n)$ times and we can ignore constant times.
With master theorem we can solve this recurrence relation and this is equal to the

$$T(n) = \theta(n \log n)$$

4. A bipartite graph whose vertices can be divided into two disjoint set U and V such that every edge connects a vertex in U to one in V. We can find that a graph is bipartite or not with coloring graph. To color graph we can use Depth First Search algorithm and it's a decrease and conquer algorithm. First, we mark every node with number 2 to indicate uncolored and also at beginning every node is unvisited. To indicate that we mark all visited status false. After that we start the depth first search algorithm. Depth First Search starts from one source vertex and goes as far as possible. First, we visit starting vertex and mark it as visited. With visiting we decreased problem size for the next iterations. And also, we give that vertex a color to checking bipartiteness of graph. After that we check the vertices that adjacent to current vertex. We start DFS process again on that adjacent vertex. Before that since to check bipartiteness of graph we switch the color for the next vertex. Process goes until we find a deadend or all neighbor are visited. Since graph may be directional we need to start DFS process for every vertex in graph. With doing that we can give color to adjacent vertexes. After that we only need to check that is there any neighbor that has same color.



If we sum all complexities:

$$\theta(V+E)+\theta(E)+\theta(V)+\theta(V)$$

We can conclude running time of algorithm is max of that summation and that is the: $\theta(V+E)$

5. We buy goods one day and we sell next day. This means our profit is daily. Because of that if we find daily profits individually, we can reach total solution from there. Let's say there is only one day to buy and one day to sell. This means day1 we buy goods and day2 we sell these goods. In day1 from cost array we can find cost of per unit of good. In day2 we sell these goods and to find selling price for per unit we look day2 from price array. If we substitute cost from price, we can find daily profit for one day. In divide and conquer approach this is actually what we do. We divide total problem into subproblems. This means for example in given PDF, we have 9 days to buy and sell. We divide days into two halves until we reach base case. When we reach base case we find profit for day1 to day2. After that we compare these two halves according to profit amounts. Maximum profit will go higher level for comparison. Also we keep a flag that show is there any day that we do profit. If there is no day that we do profit, this means we should not buy any goods in these days. No profit means, profit is equal to 0 or less than zero.

```
def isProfit(startDay, gain):
    if(gain > 0):
        return True, startDay, gain
    else:
        return False, startDay,gain

def findMaxProfitDay(cost,start,end,price):
    if(end - start == 1):
        return isProfit(start, price[end]-cost[start])
    else:
        if((end-start)%2 == 0):
            mid = ((end-start)//2)+start
            isGainLeft, leftStart,leftGain = findMaxProfitDay(cost,start,mid,price)
            isGainRight, rightStart,rightGain = findMaxProfitDay(cost,mid,end,price)
            if(leftGain > rightGain):
                return isProfit(leftStart, leftGain)
            else:
                return isProfit(rightStart, rightGain)
        else:
            mid = ((end-start)//2)+start
            isGainLeft, leftStart,leftGain = findMaxProfitDay(cost,start,mid,price)
            isGainMid, midStart,midGain = findMaxProfitDay(cost,mid,mid+1,price)
            isGainRight, rightStart,rightGain = findMaxProfitDay(cost,mid+1,end,price)
            if(leftGain>midGain and leftGain >rightGain):
                return isProfit(leftStart,leftGain)
            elif(rightGain>midGain and rightGain>leftGain):
                return isProfit(rightStart, rightGain)
            else:
                return isProfit(midStart,midGain)
```

$T(n)$

$\theta(1)$

$T(\frac{n}{2})$

$T(\frac{n}{2})$

$\theta(1)$

$T(\frac{n}{2})$

$\theta(1)$

$T(\frac{n}{2})$

$\theta(1)$

If there is one day to buy and one day to sell, this is our base case, we can easily find profit. Later, if we can divide days into two halves, we divide days with mid variable. For left part mid will be the last day to sell but since we can sell and buy same day, mid will be the start day of right part. If we can't find days into two halves equally, we should send mid day individually to base case, to provide equal size division of day. After that we compare gains with there profit amount. isProfit function finally decides if there is gain equal or less than 0, returns additionally no profit flag with False, otherwise returns True. Complexity of that algorithm

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ 2T\left(\frac{n}{2}\right) + 1 & if\ n > 1 \end{cases}$$

From master theorem complexity is $\theta(n)$.