INTRODUCTION TO ALGORITHM

HOMEWORK 4 REPORT

1) A)

Suppose:

a[s,t] + a[s+ 1, t 1] <= a[s, t + 1] + a[s + 1,t]

for all s and t such that **1 < s < n** and **1 < t < m**, and consider any i, j, k, and I sch that 1 < i < k < n and 1 j < I m. For 1 < t < m,

Sum from s=i to s=k-1 (a[s, t] + a[s + 1,t + 1]) <= Sum from s=i to s=k-1 (a[s, t + 1] + a[s + 1, t]) .

Canceling identical terms from both sides of this inequality, we obtain

a[i,t] + a[k, t + 1] a[i, t 1 + a[k, t] .

Consequently, Sum from t=j to t=l-1(a[i, t] + a[k, t + 1]) <= Sum from t=j to t=l-1 (a[i,t + 1] + a[k,t]) .

Again canceling identical terms, we obtain

a[i,j] + a[k,] < a[i, + a[k, j]

This implies A is Monge

(The Monge Array: An Abstraction and Its Applications by James Kimbrough Park)

B)

Algorithm specialArray(arr):

For from i <- 0 to i <- size
  for from j <- 0 to j<- size
    if arr[i][j]+arr[i+1][j+1] is greater than arr[i][j+1]+arr[i+1][j]
      -increment the counter
      -add arr[i][j+1] to difference between arr[i][j]+arr[i+1][j+1]
        and arr[i][j+1]+arr[i+1][j]

```
İf counter is 1:
    print the new array

İf counter is 0:
    array is already special

if counter is greater than 1 :
    array can not modified to be special with exactly 1 change
```

This algorithm checks in each iteration if the indexed elements satisfy the condition to be special array.If it is not,then the difference between them should be added in order to satisfy condition.While doing that a counter must be generated to control whether the number of change move is greater than 1 or not.The time complexity of this algortihm is **O(n^2)** because of the nested iteration.

C)

```
def getLeftMostMin(arr):
    min = arr[0]
    minIndex = 0
    for i in range(1, len(arr)):
        if(arr[i] < min):
            min = arr[i]
            minIndex = i
    return minIndex

def findMinimums(arr,start,end,index):
    if (start == end):
        index[start] = getLeftMostMin(arr[start])
    else:
        if((end-start)%2==0):
            mid = (end-start)//2
            findMinimums(arr, start, mid-1, index)
            findMinimums(arr, mid, mid,index)
            findMinimums(arr, mid+1, end, index)
        else:
            mid = ((end-start)//2)+start
            findMinimums(arr, start, mid, index)
            findMinimums(arr, mid+1, end, index)

def minRow(arr):
    index = []
    for i in range(len(arr)):
        index.append(0)
    findMinimums(arr, 0, len(arr)-1, index)
    return index
```

D)

T(m,n) = 2T(m/2)+O(n-1)

2)

Approach to solution is below:

If one of the given arrays is empty,then kth element in the non-empty array is result.

Otherwise take the middle index of the each array,if mid1 + mid2 <k then the right part of one of the array should be checked.
If mid1th element of array1 is greater then mid2th element of array2,then right part of the array2 should be recursed with the search number k-mid2-1 and vice versa.
If mid1+mid2 >k ,then left part of one of the array should be checked in the same way.

```
def kthlargest(arr1, arr2, k):
    if len(arr1) == 0:
        return arr2[k]
    elif len(arr2) == 0:
        return arr1[k]

    mida1 = int(len(arr1)/2)
    mida2 = int(len(arr2)/2)
    if mida1+mida2<k:
        if arr1[mida1]>arr2[mida2]:
            return kthlargest(arr1, arr2[mida2+1:], k-mida2-1)
        else:
            return kthlargest(arr1[mida1+1:], arr2, k-mida1-1)
    else:
        if arr1[mida1]>arr2[mida2]:
            return kthlargest(arr1[:mida1], arr2, k)
        else:
            return kthlargest(arr1, arr2[:mida2], k)
```

In the worst case kth element is the last or first element in any array.If the array sizes are m and n,then the time complexity is O(logn+logm) for the worst case.

3)

```
def maxCrossingSum(arr, l, m, h) :
    sm = 0; left_sum = -10000
    for i in range(m, l-1, -1) :
        sm = sm + arr[i]
        if (sm > left_sum) :
            left_sum = sm
    sm = 0; right_sum = -1000
    for i in range(m + 1, h + 1) :
        sm = sm + arr[i]
        if (sm > right_sum) :
            right_sum = sm
    return left_sum + right_sum
def maxSubArraySumHelper(arr, l, h) :
    if (l == h) :
        return arr[l]
    m = (l + h) // 2
    return max(maxSubArraySumHelper(arr, l, m), maxSubArraySumHelper(arr, m+1, h), maxCrossingSum(arr, l, m, h))
def maxSubArraySum(arr):
   n = len(arr)
   return maxSubArraySumHelper(arr, 0, n-1)
```

Approach to solution is below:

Divide the array in two halves in order to make calculations and decide.

a) Call the maximumSubArrayHelper for the left half (Make a recursive call)

b) Call the maximumSubArrayHelper for the right half (Make a recursive call)

c) Calculate the maximum subarray sum such that the subarray crosses the midpoint

Return maximum of the a,b,c

Recurrence relation:

$T(n) = 2T(n/2) + \Theta(n)$
Run time complexity of this algorithm is $\Theta(n\log n)$.

4)

```
def colorGraph(G, color, pos, c):

    if color[pos] != -1 and color[pos] != c:
        return False
    color[pos] = c
    ans = True
    for i in range(0, len(G)):
        if G[pos][i]:
            if color[i] == -1:
                ans &= colorGraph(G, color, i, 1-c)

            if color[i] !=-1 and color[i] != 1-c:
                return False
        if not ans:
            return False
    return True
def isBipartite(G):
    color = [-1] * len(G)
    pos = 0
    return colorGraph(G, color, pos, 1)
```

This algorithm uses adjacency matrix.Firstly initialize color array such that each element is -1.Make a call with start vertex.
Then for each vertex adjacent to that vertex,if it is not visited make recursive call.

The runnig time of the worst case is O(E+V) which E means number of edges and V means number of vertices.

5)

```
def optimalDayHelper(A,B,index):
    if (len(A) == 1):
        return [B[0]-A[0],index]
    if (len(A) > 1):
        k = len(A)//2
        x = optimalDayHelper(A[0:k],B[0:k],k)
        y = optimalDayHelper(A[k:2*k],B[k:2*k],2*k)
        return compareAndMerge(x,y)
def compareAndMerge(x,y):
    if x[0] < y[0]:
        return y
    else:
        return x
def optimalDay(A,B):
    A = A[:-1]
    B = B[1:]
    size = len(A)
    tempA = A
    tempB = B
    flag = False
    if len(A)%2 == 1:
        flag = True
        A = A[:-1]
        B = B[:-1]
    result = optimalDayHelper(A,B,len(A))
    if result[0] < 1:
        print("There is no day to make money")
        return [0,0]
    if flag == True:
        if (tempB[size-1]-tempA[size-1]) > result[0]:
            return [(tempB[size-1]-tempA[size-1]),size]
    return result
```

Approach to solution is below:

First of all input arrays are not comparable because of the None elements in each array this function clear the None elements and copy the arrays to temp.(In case of array length is odd there will be remove for the last element and after calculating best day fort he remaining,last day will be compared explicitly)

For each iteration,there is a recursive call for the left and right half of the array until reaching size=1.

X and Y variables keeps the 2 elements array,first one is the gain of that day and second is index of that day to keep.On each recursive call,the bigger gain is returned with its index with compareAndMerge.Although this

algortihm is divide and conquer,it has still linear time(O(n)) for every cases because of the calculation for gain of each day.