= CSE 321 Homework 4 =

1)

a) Given base case is: $k = i+1$. We will use induction to prove this question

Assume that, we will prove for $k = i+n$ and $k+1 = i+n+1$.

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

Let substitute $i+1 = k$

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

if we assume $k = i$, then $k+1 = i+1$ and $k = k+1$;

2) $A[k,j] + A[k+1,j+1] \leq A[k,j+1] + A[k+1,j]$

Add 1 and 2;

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

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

we assumed that $k=i$ and so $i=k$ and $i+1 = k+1$;

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

b)

```
procedure check_array (arr[0,..., m-1, 0,..., n-1], m, n, difference[0,..., k])
    newArr[0,..., 4l-1]
    index = 0
    index_dif = 0
    for i=0 to m-2 do
        for j=0 to n-2 do
            if arr[i,j] + arr[i+1,j+1] > arr[i,j+1] + arr[i+1,j] do
                newArr[index] = i
                newArr[index+1] = i+1
                newArr[index+2] = j
                newArr[index+3] = j+1
                index = index + 4
                difference[index_diff] = (arr[i,j] + arr[i+1,j+1]) - (arr[i,j+1] + arr[i+1,j])
                index_dif = index_dif + 1
            end if
        end for
    end for
end
```

* This procedure checks if given array is special or not respect formula.
If array is not special, one of 2 possibilities will be chosen.
1) One of the big ones will be reduced by difference.
2) One of the smaller ones will be increased by difference.
Stored 4 indexes which distrupt the special status in newArr and
array difference stores the differences between numbers that do not
meet the condition.

```
procedure change_array ( arr[0,...,m-1, 0,...,n-1], subarray[0,...,4(-1],difference[0,...,k])
    length = 4(
    temp =0
    while difference do
        i=0
        diff = difference [i/4]
        if temp == 0 do
            arr [subarray[i+1], subarray[i+2]] = arr [subarray[i+1], subarray[i+2]] +diff
            temp = 1
        else if temp== 1 do
            arr[subarray[i], subarray[i+3]] = arr [subarray[i], subarray[i+3]] + diff
            temp = 2
        else if temp == 2 do
            arr[subarray[i],subarray[i+2]] = arr [subarray[i], subarray[i+2]] -diff
            temp =3
        else if temp == 3 do
            arr [subarray[i+1],subarray[i+3]] = arr [subarray[i+1], subarray[i+3]] - diff
            temp =0
        end if
        i=i+4
    end while
    difference = [0,...,k]
    last = check-array (arr, len (arr) , len (arr[0]) , difference)
    if difference do
        if temp ==0 do
            arr [subarray[i+1], subarray[i+3]] = arr [subarray[i+1], subarray[i+3]] + diff
        else if temp == 1 do
            arr [subarray[i+1], subarray[i+2]] = arr [subarray[i+1], subarray[i+2]] - diff
```

```
            else if temp == 2 do
                arr[subarray[i], subarray[i+1]] = arr[subarray[i], subarray[i+1]] - diff
            else if temp == 3 do
                arr[subarray[i], subarray[i+2]] = arr[subarray[i], subarray[i+2]] + diff
            end if
        else
            break
        end if
        difference.pop(0)
        last = check_array(arr, len(arr), len(arr[0]), difference)
    end while
end
```

* As long as, the difference array from the previous procedure is full, unless the given array is not special, it will be processed for situations that distrupt the special status case.

1) One of the smaller ones will be increased by difference.
2) The other of the smaller ones will be increased by difference.
3) One of the big ones will be reduced by difference.
4) The other of the big ones will be increased by difference.

One of these options will be selected and temp value will also indicate another condition in case this selected state does not work. After this happens, the array is sent to check_array procedure for rechecking. If difference still exists, the transaction is rolled back and the next transaction is attempted. When the correct process is found, the loops ends and the new array is formed and goes driver.

c) find_leftmost → checks each row of the array one by one, makes comparisons, and finds the leftmost minimum element along the arriving array.

find_leftmost_minelement → function which called by driver, divides array into two half, left and right, respect its row indexes.

a) We assumed that : $i = 2k+1$, $k \geq 0$ so, $index_{i-1} \leq index_i \leq index_{i+1}$

Then, at most $\frac{index_{i+1} - index_{i-1} + 1}{k} = index_i$ $\quad 2k \rightarrow index_{2i+2} - index_{2i-2}$

$$T(m,n) = \sum_{i=0}^{m/2-1} (index_{2i+2} - index_{2i-2} + 1)$$

$$= \sum_{i=0}^{m/2-1} index_{2i+2} - \sum_{i=0}^{m/2-1} index_{2i-2} + \underbrace{1 + \cdots + 1}_{m/2 \text{ times}} \rightarrow m/2$$

$$= \sum_{i=0}^{m/2-1} index_{2i+2} - \sum_{i=0}^{m/2-1} index_{2i-2} + m/2$$

$$= \underbrace{index_m - index_0} + m/2$$

$$= n + m/2 \longrightarrow \underline{O(m+n)} //$$

2) Base cases are :

* if array B is empty, kth element is in array A.
* if array A is empty, kth element is in array A.
* if total size of arrays A and B is smaller than kth element, prints an error message and returns None
* if k is smaller than 1, then existance of kth element is impassible.

After this control statements, calculates middle points to divide this arrays. If summation of middle points of array A and B are smaller than k,

* if middleth element A is bigger than middle element of B, than call the same function with A, divided B, and decreased k by starting point of array B.
* if middleth element of A is equal to or bigger than middle element of B, then call the same function with B, divided array A, and decreased k by starting point of array A.

If sum of middleth points of arrays A and B are equal to or bigger than k,

* if middeth element of A is bigger than middleth element of B, then call the same function with B, divided A which is ending with middleth element and k.
* if middleth element of array A is equal to or bigger than middleth element of array B, than call the same function with A, divided B which is ending with middleth element and k.

Worst case analyze :

This function has if, elif, else statements which complexity is O(1), recursive calls at the same time : This type of tail recursion has $\underline{O(m+n)}$ complexity. //

3) * maxsum - subarray :

Base case is: if size of array arr is 1, returns the element.

After the control statement, calculates middle index of array arr and summation of left half and right half with recursive call by divide array, then sum of this array with call calculate-sum function. Then, to find max sum it is necessary to compare right half sum, left half sum and sum of the array. max sum of these sums is the result, returns the max.

* calculate_sum :

Keeps middleth element, sums from first index to middle index and the sum is bigger than left half, then update left half with sum which is made.

keeps middle+1 th element, sums from this index to last index and if sum is bigger than right half, then update right half with sum which is made, returns sum of left half and right half.

* find-array

iterates every element of array to find subarray which has max sum.

Worst case complexity :

maxsum-subarray → divides 2 half and makes call → $2 \cdot T(n/2)$

calculate-sum → has for loop → $O(n)$

find-array → has nested for loop → $O(n^2)$

$$T(n) = 2 \cdot T(n/2) + \underbrace{\Theta(n) + \Theta(n^2)}_{n^2}$$

$a = 2$

$b = 2$

$d = 2$

$2 < 2^2$ → $\Theta(n^2)$

4) * isBipartite :

first of all colored array colored default. (-1)
first vertex colored with 1 and their adjacents will color with 0.
temp keeps adjacency index which is has value 1 of array and removes one by one, checks if the vertex adjacent with own, if this condition true, returns false. then calls coloring function and if this function returns false, this function returns false, too, else returns true.

* coloring :

Iterates all the elements of array and checks if the element of array is 1, so, if the vertex has an adjacent or not, and colored with some index which is adjacent of vertex is not colored, then if the vertex is colored with 0, then color its adjacent with 1 or if the vertex is colored with 1, then color its adjacent with 0. Appends the index of adjacent of vertex to temp, if the vertex has an adjacent and its and its adjacent have same colors, return false.

Worst case complexity : (n is vertex number)

coloring → O(n)
isBipartite → O(n*n)
} O(n2)

5) * bestday :

calculates gain for per day and finds bestday respect to array of gain, and returns index of best day.

* calculate_bestday :

calls bestday function with left half and right half, calculates best days for two half and returns best day.
Especially, first of all, checks if there is no day to make money, so, if cost day is 1, prints this information on screen and returns -1 to make difference between else condition. Reports days which could not make money, too.

Worst case complexity :

bestday → has for loop → O(n)
calculate_bestday → O(n*1)
} O(n)