CSE321 Introduction to Algorithm Design

Homework4 Report

Part1

Part1 a)

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

We know that $k > i$. In the best case k can be higher only 1. Let $k = i+1$. We know that it is true. If we assume that $k = i+n$, then we prove $k+1 = i+n+1$. If it is true, then the conditation is proved.

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

$k = i+1$

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

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

Let sum ① and ②.

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

$A[i, j+1] + \cancel{A[k, j]} + A[k+1, j] + \cancel{A[k, j+1]}$

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

This equation is true, so that it was proved.

Part1 b)

Pseude Code:

```
Function convertsIntoTheSpecialArray(array[0:m][0:n])
        AllErrors = isSpecialOrNot(array)
        If AllEerrors is empty then
                Print “It is already a Special Array”
                return ERROR
        end if
        errorOf_AllErrors = findMaxErrorIndex(AllErrors)
        while (true) do
                while (i < size(errofOf_AllErrors)) do
                        tempArray = array
                        x = errorOf_AllErrors[i][0]
                        y = errorOf_AllErrors[i][1]
                        tempArray[x][y] = tempArray[x][y] + errorOf_AllErrors[0]
                        flag = isSpecialOrNot(tempArray)
                        if flag is empty then
                                return tempArray
                        end if
                        tempArray = array
                        tempArray[x][y] = tempArray[x][y] - errorOf_AllErrors[0]
                        if flag is empty then
                                return tempArray
                        end if
                end while
        end while
end function
```

```
Function isSpecialOrNot(array[0:m][0:n])
        row = size(array[0])
        column = size(array)
        for i = 0 to column -1 do
                for j = i + 1 to column do
                        for k = 0 to row -1 do
                                for l = k + 1 to row do
                                        upper_left = array[i][k]
                                        upper_right = array[i][l]
                                        lower_left = array[j][k]
                                        lower_right = array[j][l]
                                        if upper_left + lower_right > lower_left + upper_right then
                                                errorCoordinateAndDifference =
                                                        [first - second, [i,k], [i, l], [j, k], [j, l]]
                                                errorArray.insert(errorCoordinateAndDifference)
                                        end if
                                end for
                        end for
                end for
        end for
        return errorArray
end function
Funtion findMaxErrorIndex(array[0:x])
        // return the max error difference. The difference is first – second in the isSpecialOrNot
        function. Basic find max item in the given array problem
```

Explanation of the Algorithm:

To solve this problem, I implemented 3 major methods. **convertsIntoTheSpecialArray, isSpecialOrNot** and **findMaxErrorIndex.** isSpecialOrNot method checks that if the 2D Array is Special Array or not. If it is not Special Array, inserts the difference amount and the indexes which break the condition of the Special Array, into an array. convertsIntoTheSpecialArray method converts the 2D Array into the Special Array. (It must be only one item to change). findMaxErrorIndex method finds and returns the index of the maximum difference amount in an array.

I call convertsIntoTheSpecialArray with a 2D Array. Inside of it I call isSpecialOrNot method. isSpecialOrNot method returns an array which includes difference amount and indexes. I must find maximum difference, which is inside in the returning array of isSpecialArray. Problem can be solved with only maximum number of difference. Then I create a loop with infinity. Inside of this loop I create a new loop with size 4. Meaning of 4 is there is 4 point (upper_left, lower_left, upper_right, lower_right). Then I add and take out the difference for each point, then I check the new 2D Array with isSpecialOrNot method. If it is Special Array, returns the new array and indexes of changing element indexes, otherwise keep going with the loop until find the convert it to Special Array

Part1 c)

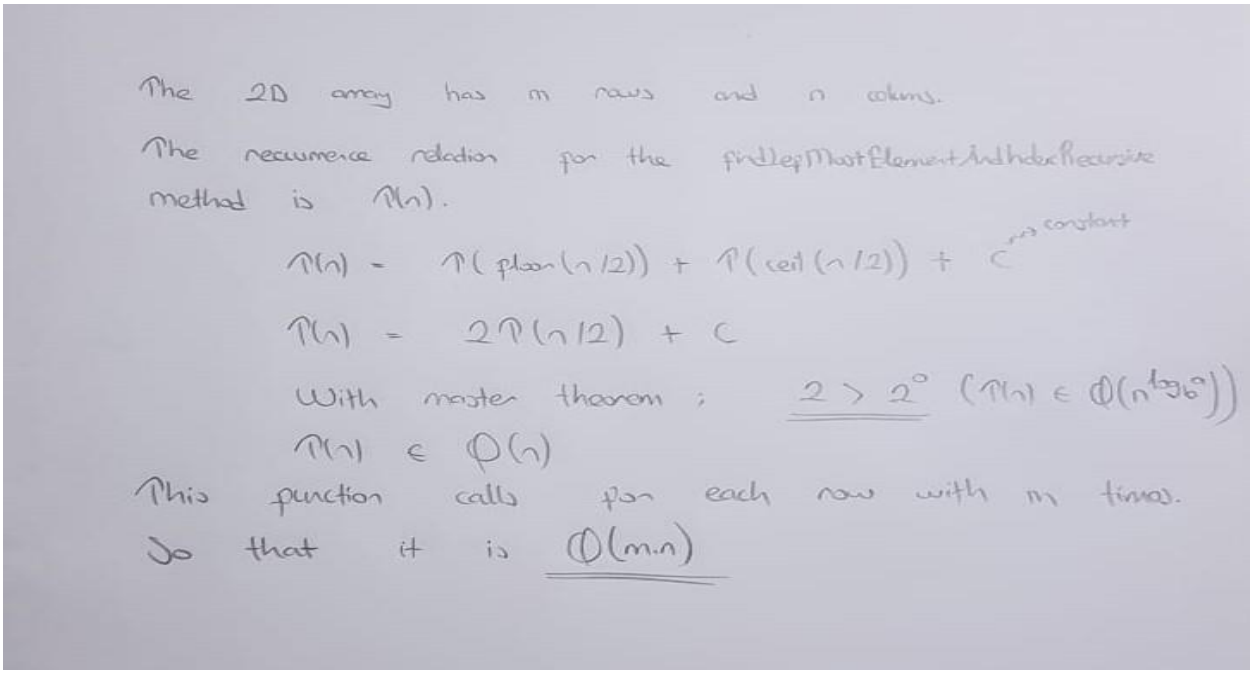
Explanation of the Algorithm:

To solve this problem, I implemented **findLeftMostMinumumElementAndIndex**

and **findLeftMostMinumumElementAndIndexRecursive** methods. Inside of the findLeftMostMinumumElementAndIndex method I call findLeftMostMinumumElement AndIndexRecursive method for each row in 2D Array (Special Array). This method is a divide and conquer algorithm to find minimum element and its index.

The best case of this divide and conquer algorithm is being equal to start index and end index for the given array. If these are equal, returns the index and its value of inside the array. Otherwise, it finds the middle index, and recalls the method for the left part of the array until its start and end indexes are equal. Then recalls the method for the right part of the array until its start and end indexes are equal. Finally, I can know the minimum number and its index for left and right part of the array, then it makes a comparison to find minimum of these. Then returns the minimum element and its index.

Part1 d)



Part2

To solve this problem, I implemented 2 methods. These are **find_kthElemet** and **find_kthElementRecursive.** find_kthElement just calls find_kthElementRecursive method with arrays, size of arrays and k (searching index of element). find_kthElementRecursive method has 4 best cases. Let assume that the size of arrays are m and n

-If k bigger than m + n or smaller than 0 (Error Case)

-If m equals 0, otherwise array1 is empty (searching item is in the array2 on k -1 index)

-If n equals 0, otherwise array2 is empty (searching item is in the array1 on k -1 index)

-If k equals 1 (searching item is the min item of array1[0] and array2[0])

Divide and conquer part is:

Divide k by 2. Not m or n.

Create index1 and index2 with minimum value of (m and k/2) and (n and k / 2)

If the item of array1 in index1 is bigger than the item of array2 in index2, then the algorithm focuses on the array2 with from index2 to n. In this case searching item is (k-index2)th item.

Otherwise the item of array1 in index1 is smaller than the item of array2 in index2, then algorithm focuses on the array1 with from index1 to m. In this case searching item is (k-index1)th item.

In other words, I compare both arrays (k / 2)th item, not the middle item of the arrays. When (k / 2) reaches the 1 the item finds.

The time complexity of this algorithm is O(log(k)), because k divides by 2 in each iteration until k becomes 1.

Worst-case of the algorithm occurs to find item in (m + n) index. In this case the time complexity is **O (log (m + n)).**

Part3

To solve this problem, I implemented **findContiguousSubsetWithLargestSum**, **findContiguousSubsetWithLargestSumRecursive** and **findMaxCrossingSubArraySum** methods.

findContiguousSubsetWithLargestSum methods calls findContiguousSubsetWithLargestSumRecursive and returns result subset and sum of it. findContiguousSubsetWithLargestSumRecursive method is a divide and conquer method.

The best case of it, having 1 item in the array. In this case it returns the indexes and the value of the array.

Otherwise, the maximum subset can be in the left half, right half, or cross the middle of the array. So that it calculates middle index of the array and recalls this method with left part of the array and the right part of the array. Recursive method returns subset of the left side and the right side. But we need to find maximum subset. So that the recursive method calls findMaxCrossingSubArraySum method. It sums the left and right part of the middle item in each call. And it returns the sum, and indexes of start and end. Then, there are 3 subsets, these are left part of the array, right part of the array and the cross the middle of the array. Finally, the recursive method checks which subset is maximum and returns its sum value and indexes.

$T(n) = 2T(n / 2) + \Theta(n)$. {2T (n / 2) is recursive call, Θ (n) is sum of left and right part of middle item}

$T(n) = \Theta(1)$. {if n = 1, best case}

Solving the recurrence relation with Master's Theorem:

$a = 2, b = 2, d = 1$.

$a = b^d$ so that $T(n) = \Theta(n*\log(n))$.

Worst-case is Θ (n*log(n)).

Part4

To solve this problem, I created I class with name Graph. I choose adjacency matrix method to implement Graph class. There is a 2D array which represent the graph, a colorArray (I will mention) and vertex size. In the constructor of the class I give their initial values. And I implemented **isBipartiteGraph** and **isBipartiteGraphBFS** methods. I used Breadth-First Search algorithm inside of the isBipartiteGraphBFS method to make it decrease and conquer. We know that the meaning of Bipartite Graph. Let there are Set U and Set V, each edge must be starts from Set U and ends in Set V or starts from Set V and ends in Set U. To make it simple I create some colors. These colors are RED and BLACK. RED means that the vertex is in the one set. BLUE means that the vertex is in another set. In the constructor of the Graph I gave RED to all indexes of colorArray.

In isBipartiteGraph method, I call isBipartiteGraphBFS method with graph and index which is the RED of colorArray. If isBipartiteGraphBFS returns false then the graph is not bipartite, otherwise keep going call isBipartiteGraphBFS method with different index which is the RED of colorArray.

Inside the isBipartiteGraphBFS there is a basic BFS algorithm. And also checking the color. I created a visited array and insert the index which is the parameter. With a loop, which runs the visited array becomes empty, check the color. If there is an edge from U to V and its color is RED, assign BLACK to edge of V to U. Otherwise, if there is an edge from U to V and their color are the same return false because 2 edge has the same color and it makes the Graph not bipartite.

In the worst case I have to control all edges. I choose adjacency matrix method so that **Worst-case is Θ (V^2).** V means that the number of vertexes.

!!! IMPORTANT !!!

My Graph initialization is like that:

[0, 1, 1, 0],
[1, 0, 0, 1],
[1, 0, 0, 1],
[0, 1, 1, 0]

If you want to test this part, you have to give a vertex number when creating new Graph object and you have to represent the graph as shown above.

This means that:

0 -> 1 -> 2 | 1 -> 0 -> 3 | 2 -> 0 -> 3 | 3 -> 1 -> 2

Part5

To solve this problem, I implemented **findBestDayToBuyGood** and **findBestDayToBuyGoodRecursive** methods. findBestDayToBuyGood methods takes 2 arrays which represent the Cost and Price of days. This method calls findBestDayToBuyGoodRecursive with start, end parameter of the arrays and Cost and Price arrays. It returns the best day to buy good and its gain. findBestDayToBuyGoodRecursive method is a divide and conquer algorithm to find maximum gain element and its day value.

The best case of this divide and conquer algorithm is being equal to start and end. If these are equal, it returns the day which is "start + 1" and gain which is "Pain – Cost". Otherwise, it finds the middle index, and recalls the method for the left part of the arrays until its start and end indexes are equal. Then recalls the method for the right part of the array until its start and end indexes are equal. Finally, I can know the gain numbers and its days value for left and right part of the arrays, then it makes a comparison to find maximum gain of left and right parts. Then returns the maximum gain and its day value.

$T(n) = 2T(n / 2) + c$ {2T (n / 2) is recursive calls, c is constant operations}

$T(n) = \Theta(1)$. {if start equals to end}

Solving the recurrence relation with Master's Theorem:

$a = 2, b = 2, d = 0$.

$a > b^d$ so that $T(n) = \Theta(n^{\log_b a})$.

$T(n)$ is $\Theta(n)$.

In worst case it must compare all items so that **$W(n)$ is $\Theta(n)$.**

!!! IMPORTANT !!!

In Cost array the last index cannot be known. So that when you create an array for Cost, you must give "**NoBuy**" for the last index.

In Price array the first index cannot be known. So that when you create an array for Price, you must give "**NoSale**" for the first index.

For example:

Cost = [5, 11, 2, 21, 5, 7, 8, 12, 13, NoBuy]
Price = [NoSale, 7, 9, 5, 21, 7, 13, 10, 14, 20]

Expected Outputs of Given Driver Methods

```
================================================================================
Algorithm HW4 Part1_b
Your 2D Array is
[10, 17, 13, 28, 23]
[17, 22, 16, 29, 23]
[24, 28, 22, 34, 24]
[11, 13, 66, 17, 7]
[45, 44, 32, 37, 23]
[36, 33, 19, 21, 6]
[75, 66, 51, 53, 34]
After chancing one single element
Changing Row : 3 Changing Column : 2
[10, 17, 13, 28, 23]
[17, 22, 16, 29, 23]
[24, 28, 22, 34, 24]
[11, 13, 7, 17, 7]
[45, 44, 32, 37, 23]
[36, 33, 19, 21, 6]
[75, 66, 51, 53, 34]
================================================================================
================================================================================
Algorithm HW4 Part1_c
Your 2D Array is
[10, 17, 13, 28, 23]
[17, 22, 16, 29, 23]
[24, 28, 22, 34, 24]
[11, 13, 6, 17, 7]
[45, 44, 32, 37, 23]
[36, 33, 19, 21, 6]
[75, 66, 51, 53, 34]
After Finding Leftmost Minimum Element and Its Index in Each Row
Value : 10 Index : [0,0]
Value : 16 Index : [1,2]
Value : 22 Index : [2,2]
Value : 6 Index : [3,2]
Value : 23 Index : [4,4]
Value : 6 Index : [5,4]
Value : 34 Index : [6,4]
================================================================================
```

```
================================================================================
Algorithm HW4 Part2
Your Sorted Array 1 : [5, 10, 15, 20, 25]
Your Sorted Array 2 : [3, 6, 9, 12]
4th Element : 9
================================================================================
================================================================================
Algorithm HW4 Part3
Your Array : [5, -6, 6, 7, -6, 7, -4, 3]
Contiguous Subset With Largest Sum : [6, 7, -6, 7]
Sum : 14
================================================================================
================================================================================
Algorithm HW4 Part4
Your Adjacency Matrix Graph is
[0, 1, 1, 0]
[1, 0, 0, 1]
[1, 0, 0, 1]
[0, 1, 1, 0]
0 -> 1 2
1 -> 0 3
2 -> 0 3
3 -> 1 2
The Graph is Bipartite
Your Adjacency Matrix Graph is
[0, 1, 0, 1]
[1, 0, 1, 0]
[0, 1, 0, 1]
[1, 0, 1, 1]
0 -> 1 3
1 -> 0 2
2 -> 1 3
3 -> 0 2 3
The Graph is not Bipartite
================================================================================
```

```
================================================================================
Algorithm HW4 Part5
Your Cost is
[5, 11, 2, 21, 5, 7, 8, 12, 13, No Buy]
Your Price is
[No Sale, 7, 9, 5, 21, 7, 13, 10, 14, 20]
The best day to buy goods is the 9th day
The gain is 7
================================================================================
================================================================================
151044058 ALİ HAYDAR KURBAN
================================================================================

Process finished with exit code 0
```

ALİ HAYDAR KURBAN 151044058