CSE 321
INTRODUCTION TO ALGORITHM DESIGN

HOMEWORK 04 REPORT

Ömer ÇEVİK

161044004

1 Question's Solution

In this part,

a) In the definition: " An m x n array A of real numbers is called a special array if for all i, j, k and l such that $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$, we have: A[i, j] + A[k, l] $\leq$ A[i, l] + A[k, j] "

To prove that i = 1, 2, ... , m - 1 and j = 1, 2, ... , n - 1, we have:
A[i, j] + A[i + 1, j + 1] $\leq$ A[i, j + 1] + A[i + 1, j]

Using Induction Proof:

⊛ For i = 1 and j = 1 then A[i, j] + A[i + 1, j + 1] $\leq$ A[i, j + 1] + A[i + 1, j] becomes to A[1, 1] + A[2, 2] $\leq$ A[1, 2] + A[2, 1].

In special array definition $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$ is provided for i = 1 < k = 2 and j = 1 < l = 2. Correct ! ✓

⊛ For i = I and j = J then assume that A[I, J] + A[I + 1, J + 1] $\leq$ A[I, J + 1] + A[I + 1, J] is correct. ✓

⊛ For i = I + 1 and j = J + 1 then A[I + 1, J + 1] + A[I + 1 + 1, J + 1 + 1] $\leq$ A[I + 1, J + 1 + 1] + A[I + 1 + 1, J + 1] and it equals to A[I + 1, J + 1] + A[I + 2, J + 2] $\leq$ A[I + 1, J + 2] + A[I + 2, J + 1].

It seems that the rule of $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$ correct for it. The formula is proved correctly!

Proved! ✓

b) **The Pseudo Code :**

```
for i in range(1, len(A)):
    for j in range(1, len(A[i])):
        for k in range(i+1, len(A)):
            for l in range(j+1, len(A[i])):
                if not A[i][j] + A[k][l] ≤ A[i][l] + A[k][j]:
                    return element and its index as an array
```

In that algorithm, given the rule is followed as indexes i, j, k and l. This rule is given : $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$ and the loops in that algorithm designed to that rule.

In that algorithm, given the rule is followed as indexes i, j, k and l. This rule is given : The 'm' is size of columns in A array: len(A). The 'n' is size of rows in A array : len(A[i]). The condition of A[i][j] + A[k][l] $\leq$ A[i][l] + A[k][j] is not applied then the result is going to be returned.

c) In this algorithm, the *Quick Sort* algorithm is implemented to sort the each row of 2D array. But there is a little difference in that *Quick Sort*. It gets an *Index* array to keep the sorting elements indexes.

Then the minimum elements in each rows are stored in zero index of the array and each index of the minimum elements stored in created *Index* array's zero indexed element.

d) **The Recurrence Relation**

$$T(n) = 2T(n^2) + O(n)$$

2 Question's Solution

In this part, user calls the *question2()* function giving parameters as A array, B array and the k^{th} element. The *question2()* function returns *findKthElement()* function which searches the k^{th} element using divide and conquer algorithm.

findKthElement() function gets A and B's right and left sub-array's lengths and also the k^{th} element. Then the most important checking mechanism is that if left sub-array length of A is smaller than right sub-array length of A and also same condition for B array is correct then evaluate the middle index of A and B. Then create the next to middle index of A and B.

⊙ The next indexes of middle indexes are larger than the k^{th} element and middle element of A is larger than B's middle element then the right sub-array length of A must be smaller than middle index of A.

⊙ The next indexes of middle indexes are larger than the k^{th} element and middle element of A is small equal than B's middle element then the right sub-array length of B must be smaller than middle index of B.

⊙ The next indexes of middle indexes are small equal than the k^{th} element and middle element of A is smaller than B's middle element then the left sub-array length of A must be larger than middle index of A and k^{th} element must be decreased by the next current middle element's of A index.

⊙ The next indexes of middle indexes are small equal than the k^{th} element and middle element of A is large equal than B's middle element then the left sub-array length of B must be larger than middle index of B and k^{th} element must be decreased by the next current middle element's of B index.

With new calculated divided arrays' lengths there is a recursive call till left sub-array length of A larger than right sub-array length of A and left sub-array length of B larger than right sub-array length of B.

⊛ Consequently this recursive call returns element in A's left sub-array length and k^{th} index's sum if left sub-array length of A small equal than right sub-array length of A **but** left sub-array length of B *not* larger than right sub-array length of B, otherwise returns element in B's left sub-array length and k^{th} index's sum.

Time Complexity :

Worst Case : This algorithm looks similar to merge sort in logically. Dividing each array till one element stays then find the element in given index. The worst case scenario almost in that recurrence relation: T(n) = T(n/2) + $\theta(1)$. Using Master Theorem we can see that is log(n).

So $T_W(n) \epsilon \underline{O(log(n))}$

3 Question's Solution

In this part, the given array is going to be parted of its contiguous sub-arrays and finding the maximum contiguous subset sum.

In my algorithm, firstly dividing the array left and right sub-arrays recursively. Then evaluating the sum of these contiguous sub-arrays using *divideAndConquerSum()* function.

While evaluating the left and right sub-arrays returning their values to compare with result of *divideAndConquerSum()* function. The maximum value of these three function call will give the maximum contiguous subset sum result.

Time Complexity :

Worst Case : Finding the left and right sub-arrays with two recursive call is : T(n) = 2T(n/2). And the *divideAndConquerSum()* function sums the given indexes almost T(n) = n. Completely we can assume that T(n) = 2T(n/2) + n. Using Master Theorem, a = 2, b = 2 and d = 1. 2 = 2^1, 2 = 2. So the result in Master Theorem will give the T(n) ϵ θ(n^dlogn). It equals to θ(n^1logn).

So $T_W(n) \epsilon \underline{\theta(nlogn)}$

4 Question's Solution

In this part, the given graph is checking out if it is bipartite or not. The bipartite graph means that when started vertice is colored to a color and its neighbors colored to distinct color and the contiguous of this operation doesn't cause to deadlock it shows that that graph is a bipartite graph.

The algorithm that written in python code creates that color array and signs the colored vertices. Decrease and conquer algorithm performs using the queue and the queue data structure is used to save the vertices indexes.

Time Complexity :

Worst Case : Time Complexity of to check bipartite graph approach is same as that Breadth First Search. In implementation is O(V^2) where V is number of vertices.

So $T_W(n) \epsilon \underline{O(V^2)}$

5 Question's Solution

In this part, user gives two arrays which one keeps cost and the other one keeps price values. Cost array gets the '-' character in the end of the array and Price array gets the '-' character in the start of array.

In my algorithm, this cost and price arrays are given to *divideAndConquerMax()* function with removed '-' characters in that arrays. That function gets the *maxGain* array to keep the result in its zero indexed element and the day of the *maxGain* is stored in one indexed element.

The *maxGain* array firstly keeps the negative infinite value in its zero and one indexed element. While comparing the subtraction of cost and price with *maxGain*, the *maxGain* updates itself with maximum value of subtraction result. That operation continuous until the all subtraction of cost and price arrays finishes. Starting point is in the middle of each this cost and prices and goes on in dividing and finding each sub-array's middle elements.

Time Complexity :

Worst Case : In that algorithm, the *divideAndConquerMax()* function has two recursive calls. It can be represented as T(n) = 2T(n/2) + $\theta(1)$.

Using the Master Theorem; a = 2, b = 2, d = 0. 2 > 2^0, 2 > 1. So result in Master Theorem will give the T(n) ϵ $\theta(n^{log_b a})$. It equals to $\theta(n^{log_2 2})$.

So $T_W(n) \epsilon \underline{\theta(n)}$