CSE 321 Instroduction to Algorithm Design

Ahmet Yuşa Telli
151044092

December 2019

1 Special Array

Part A:

In first part, we prove an array is special or not. The array has mxn size. In special array
$A[i,j] + A[k,l] \leq A[i,l] + A[k,j]$ (a)
$1 \leq i < k \leq m$ and $1 \leq j < l \leq n$ (b)

We want to prove
$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$ (c)
$i = 1, 2, ..., m-1$ and $j = 1, 2, ..., n-1$ (d)

In (a) equality, Let we say $k = i + 1$ and $l = j + 1$. We can say that using (b) equality. Then we have :
$A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$ (e)
This time, We can see new (e) equality and (c) are the same. We Proved It! If the array is true with (c) equality, We can say "It is Special Array!".

Part B:

Before find the wrong element, we check the array for special or not. In the check function $isSpacialArray(array)$ when we check the array, we take a list. In the special array rule (a) and (b), when we check this rule, if the rule is not applied, we take these indexes and push to the list.

At the end of the function and the array is not special and we have some indexes array. We check the list and find the most repetitive index. We use a dictionary for find the frequency for each index. The wrong element is which has the most frequency in the dictionary.

Psudocode:

```
loop for i,j,k,l in array:
    if A[i,j]+A[k,l]  A[i,l]+A[k,j]:
        is special cont.
    else:
        push indexes (i,j), (k,l), (i,l), (k,j) to list
    endif
endloop

for m in list:
    find repetitive index.
endfor
```

Part C:

Find the leftmost minimum element in each row. We take a two dimension special array. First we find the first row's minimum element and remove the first row then recursive call the function with new special array. At the end of the last row we go back and make a list with minimum elements and their indexes.

For example we take this array:
$[[10, 17, 13, 28, 23]$,
$[17, 22, 16, 29, 23]$,
$[24, 28, 22, 34, 24]$,
$[11, 13, 6, 17, 7]$,
$[45, 44, 32, 37, 23]$,
$[36, 33, 19, 21, 6]$,
$[75, 66, 51, 53, 34]]$

First we take the smallest element then take the index for the row. And recursive call with other row.

When we finding the minimum element in a row, we take a row and in one loop, we search the minimum element and it's index. Then, return both values.

Part D:

In Part C, we find the minimum elements in array. When we finding, we have a recursive call and for each row we have a loop. Then we find the time complexity using recurrence relation,
$T(n) \in 2 \times T(n^2) + O(n)$

2 Kth Element

We have two different array. The arrays are sorted. We try to find Kth element in these arrays.

Algorithm:

In the function, we take two arrays (A, B), their start and end indexes and K element as parameters. Our base conditions is, if start indexes are greater then end indexes for both arrays. It can not be greater. For A arrays, start index is greater than end index, the K element should be in the B array. The same way for B, the K should be in A.

First we need to find the middle elements on these arrays. Then we find Kth element is in right side or left side. We divide by two these arrays from middle we compare the middle elements. After the compare middle elements, we calculate the start and end indexes for these array. Then recursive call the function.
For example we have these arrays:
$A = [1, 3, 5, 6, 7, 8, 9, 11]$
$B = [2, 4, 10, 12, 14, 15, 17, 18, 19, 20, 21, 22]$
And the K = 19. This K must be in B array, it can not be in A.
But if K = 35. It is greater than our total arrays' size. It can not be in these arrays.

Worst-Case:

If the Kth element is greater than the arrays' sizes and it is not in arrays or Kth element is the last element of the great array. We divide by two and take right side until the last element. And the complexity is $T(n) \in O(n \log n)$

3 Largest Sum

In this part, we calculate two different things. First split the array to left and right side. Second find the total.

We split the array using $findSubArray(array, start, end)$ function:
$findSubArray(arr, left, mid)$ $findSubArray(arr, mid + 1, right)$
In base case if start index is equal to the end index, we return the left side.

We find the middle index and find all subsets on left side. Then find all subsets on right side. After subsets, find the summations for these subsets. And return the maximum number of totals.

Worst Case:

Worst case scenario is, last element is a subset and it has the maximum total. This time we need to find all subsets and compare all totals. In the $findSubArray(arr, left, right)$ function, we have two recursive call and one helper function call.

For these calls, $T(n) \in 2T(n/2) + \theta(n)$
Now we use the Master Theorem for prove, $a = 2 \quad b = 2 \quad d = 0$
Then, we find $T(n) \in \theta(n \log n)$

4 Bipartite Graph

We have a two dimension array and we find it is bipartite graph or not. To find, we need a colors array and a queue. A bipartite graph is a graph whose vertices can be divided into two disjoint and independent sets and such that every edge connects a vertex in to one in.

We color a node and it's neighbor has to take different color. Two connected nodes can not be the same color. If they are the same color, it is not a bipartite graph.

Algorithm:

In $checkBipartite(graph, colors, queue)$ function, we always check the queue. At the end of the function if queue is empty this meaning is it is bipartite. In the function, we check all "1" values. If the index is "1", check the color. If uncolored, paint the index. If colored before, check the neighbor. The neighbor has same color, it is not a bipartite graph.

We test with this 2D arrays.
[[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0]]

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

Worst Case:

In this algorithm we check all indexes in the two dimension array. We need to find all edges and vertexes. This means, best case, worst case and average case are the same result. When we checking, we find all "1" vertexes and add to queue and paint the index. Then check the neighbors. When we checking, we have a loop for number of vertexes. And we have a recursive call for number of vertex. Because we have a square array. Then we need to loop for $V \times V$

times. The time complexity is $T(n) \in O(V^2)$ (V = number of vertex).

5 Maximum Gain

We have two arrays Cost and Price. We need to find the Gain array.

Algorithm:

In the algorithm, we take cost and price arrays. Until one element, we divide these arrays. And we take one element from cost and one element from price. Then return the minus of these numbers. First we check left half the arrays then check right half. When we check, if we do not make money for a goods, we write it's cost, price and loss.

In the "$maxGain(A, B)$" function we have two recursive calls. First for is left half of arrays.
$maxGain(A[int(len(A)/2) :], B[int(len(B)/2) :])$

For the left half of arrays:
$maxGain(A[: int(len(A)/2)], B[: int(len(B)/2)])$

Worst Case:

In the worst case scenario, the maximum gain is in the last day. We start to divide from middle then go to the beginning. When we come to the last element, we go back with find the results. And we take the maximum result.

We have two recursive call with left half and right half. This means is
$T(n) = 2T(n/2) + \theta(n)$
Using the Master Theorem : $a = 2 \quad b = 2 \quad d = 0$
Then, we find $T(n) \in \theta(n \log n)$

Ahmet Yuşa Telli

151044092