CSE 321 – Introduction To Algorithm Design Homework #4

Gökçe Nur Erer – 171044079

1) a) To prove $A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$

By replacing $k = i+1$ and $l = j+1$ in the given property we get:

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

So according to that lets assume:

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

To solve this:

$A[k,j] + A[k+1,j+1] \leq A[k,j+1] + A[k+1,j]$ is given if we replace i with k.

If we sum up the assumption and the given inequalities:

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

$= A[i,j] + A[k+1,j+1] \leq A[i,j+1] + A[k+1,j]$ if we replace back k with i. The result will be $A[i,j] + A[i+1,j+1] \leq A[i,j+1] + A[i+1,j]$.

b) –

c) The algorithm designed for this question gets an array and an empty array to fill it with leftmost minimum values and leftmost minimum index tuples. First the algorithm creates two subarrays one of them containing the even rows and the other one containing the odd rows. Until any of the subarray sizes becomes 1 the algorithm calls itself recursively. When the subarray size is 1 the algorithm finds the minimum in the subarray and appends the leftmost minimum value and the indexes into the given empty array. The returned array contains all the leftmost minimum values and their row and column indexes. The first element is the value the second element is the row index and the third element is the column index of every leftmost minimum element in the returned array.

d)$T(m,n) = T(m/2,n) + O(n)$ m being row count, n being column count. $O(n)$ indicating the complexity of finding the minimum element.

2) The algorithm designed for this question gets two arrays, their start and end indices and the k value. The algorithm first checks if the arrays exist. Which is the base case which returns the k^{th} value in the other array whose does exist. Then middle indices are calculated for both of the arrays.

If the sum of these two indeces are less than k;

If the element at the middle index of the first array is greater than the element at the middle index of the second array

Algorithm is called with first array and with the subarray starting from middle of the second array to the end of the second array **(as the second array)**. And the k value is updated to k – mida2 **(middle index of the second array)** – 1.

Else,

Algorithm is called with the subarray starting from the middle of the first array to the end of the first array **(as the first array)** and with the second array. And the k value is updated to k – mida1 **(middle index of the first array)** – 1.

Else,

If the element at the middle index of the first array is greater than the element at the middle index of the second array

Algorithm is called with the subarray starting from the start of the first array and ending at the middle of the first array (as the first array) and with the second array.

Else,

Algorithm is called with the first array and with the subarray starting from the start of the second array and ending at the middle of the second array (as the second array).

The worst case of this algorithm will result in O (logm + logn). Since the arrays get divided into two halves an one is used. So this will result in log m time for an array which has length m and will result in log n time for an array which has length n. Since these operations are done seperately and don't depend on each other we sum them up to find out the complexity.

PS: Array indices start with 0 so if k^{th} element is wanted k-1 will be sent to the algorithm as a parameter.

3) This algorithm designed for this question gets an array, array's start index and end index. First the algorithm checks if the array has only one element, if so, it returns it as the maximum sum. If not the algorithm calculates the middle index of the array and calls itself recursively for its left half, right half. An helper function is used to calculate the maximum sum of any subset including the middle element. After all these functions return their values, it is checked which sum is taken as the maximum and it is returned. To keep a track on the start and end of the subarrays start and end indices are returned whenever a maximum sum is returned. To return

this subarray an helper function is used to use these start and end indices to find the subarray.

The worst case complexity of this algorithm can be calculated by writing the recurrence for the recursive part of the algorithm which is: $T(n) = 2T(n/2) + n$. When this recurrence is solved by Master's Theorem it results in $O(n*\log n)$ worst case complexity.

4) The algorithm designed for this question gets an adjacency list, a list that shows if the nodes are visited, and a list showing which set the node belongs to. Algorithm first traverses all graph using DFS and marks all nodes with a set label. This set label can be either 0 or 1 and depends on the neighbour's set. The opposite set of the neighbour is given to the current visited node.

Then it checks if any adjacent nodes have the same set label or not. And depending on that algorithm decides if the graph is bipartite or not.

The worst case of this algorithm results in the same worst case complexity of DFS using adjacency lists which is $O(N+E)$ which N stands for node count and E stands for edge count. Because for each node, you can discover all its neighbors by traversing the adjacency list of the node just once in linear time. For an undirected graph, each edge will appear twice. The sum of the sizes of the adjacency lists of all the nodes is E. So, the overall complexity will be $O(N) + O(2E)$ which is $O(N+2E)$ which can be represented as $O(N+E)$.

PS: Nodes are indexed with 0 to … n-1, adjacency list given according to that.

5) The algorithm designed for this question gets the array of the costs of each day and prices of each day. The elements of the array can either be '-' to indicate that non-exists or numbers. First the algorithm calculates the gains for each day, then by using a helper function which has a divide and conquer approach finds the maximum gain in the gain array that has been created. To find the maximum gain if one element left the maximum is that one element. If there are two elements left the greater value is the maximum. For the other cases the array is divided in to two and this algorithm is called recursively. Lastly, a comparison is made between the maximum of the left half of the array and the maximum of the right half of the array. And depending on the result the maximum is found. Also, to find the day where the maximum gain calculated the index value of the maximum gain returned.

When the maximum gain value and the day of the maximum gain returns to the function it is checked that if the maximum gain is negative or zero so can inform that there is no point making any sales. If the gain is positive, then the maximum gain and the day to sell to get that gain is printed to the screen.

The worst case of this algorithm is $O(n)$, n indicating the day count, costs array size, prices array size. By writing the recurrence relation of this algorithm for finding the

maximum which is: $T(n) = 2*T(n/2) + 1$ and by solving it with Master's Theorem, the number of comparisons is n, yet the worst case is $O(n)$.