Gökhan HAS
161044067

CSE 321 HW-04

1-) SPECIAL ARRAY:

a) According to the definition of special array's the "only if" part is trivial. As for the "if" part, let's prove that:

$$A[i,j] + A[i+1,j+1] \le A[i,j+1] + A[i+1,j]$$
$$A[i,j] + A[k,j+1] \le A[i,j+1] + A[k,j] \quad \text{where } i<k$$

The base case of $k = i+1$ is given. We can use prove by induction method, As for the inductive step, we assume it holds for $k = i+n$ and we want to prove it for $k+1 = i+n+1$. If we add the given to the assumption:

$$A[i,j] + A[k,j+1] \le A[i,j+1] + A[k,j] \quad \text{(assumption)}$$
$$A[k,j] + A[k+1,j+1] \le A[k,j+1] + A[k+1,j] \quad \text{(given)}$$
$$A[i,j] + A[k,j+1] + A[k,j] + A[k+1,j+1] \le A[i,j+1] + A[k,j] + A[k,j+1] + A[k+1,j]$$

So, $A[i,j] + A[k+1,j+1] \le A[i,j+1] + A[k+1,j]$

b)
```
Algorithm convertSpecial (A[1...m][1...n])
    for i ← 0  m-1 to do
        for j ← 0  n-1 to do
            if $A[i][j] + A[i+1][j+1] > A[i][j+1] + A[i+1][j]$
                $A[i][j+1] \leftarrow A[i][j+1] + (A[i][j] + A[i+1][j+1]) - (A[i][j+1] + A[i+1][j])$
            end if
        end for
    end for
```

I wrote this pseudocode based on special array definition. The loops are used to iterate for the 2D array. Using the "If" condition detects the case that breaks the special array. Then, that element is provided to be equal to the smallest expression, satisfying the condition.

```
        ↓ A[i][j]
Ex: ... 34   33 ...
    ...  4    5 ...
         ⋮    ⋮
```

$$\overset{(39 \;>\; 37)}{\text{if } 34+5 > 33+4}$$

$$A[i][j+1] = 33 + (39-37) \rightarrow 35$$

New Array is $\rightarrow$

```
34  35
 4   5
```

Then, satisfying the special array condition.

c) In this question, since the divide and conquer algorithm was requested, I found it appropriate to use merge sort. Merge sort is a divide and conquer algorithm.

```
MergeSort (arr, left, right)
    If left ≤ right
        1- Find the middle point to divide the array into two halves:
              middle m = (left + right) / 2
        2- Call mergesort for first half:
              merge Sort (arr, left, m)
        3- Call mergesort for second half:
              merge Sort (arr, m+1, right)
        4- Merge the two halves sorted in step 2 and 3:
              merge (arr, left, m, right)
```

I send all lines to mergeSort function and add the first element of each line to the new array. (line = row)

d) The divide time is $O(1)$, the conquer part is $T(m/2)$ and the merge part is $O(m+n)$.

$$T(m) = T(m/2) + cn + dm$$
$$= cn + dm + cn + d\frac{m}{2} + cn + d\frac{m}{4} \dots$$
$$= \sum_{i=0}^{\lg m-1} cn + \sum_{i=0}^{\lg m-1} \frac{dm}{2^i}$$
$$= cn\log m + dm \sum_{i=0}^{\log m-1}$$
$$< cn\log m + 2dm \qquad \in O(n\log m + m)$$

2-) FINDS THE K^TH ELEMENT

- I compare the middle elements of arrays array1 and array2. I said that these indices are mid1 and mid2. Assume that array1[mid1] is equal to k, then clearly the elements after mid2 cannot be the required element. Then, set the last element of array2 to be array2[mid2].
- Let us assume that arrays are A and B. The inputs are right. k's range is [0, len(A) + len(B)]. If length of one of the arrays is 0, the answer is k^th element of the second array. This is the base case of divide and conquer algorithm.
- If mid index of A + mid index of B is less than k
  → If mid element of A is greater than mid element of b, we can ignore the first half of b; else ignore the first half of A, adjust k.
- Else if k is less than sum of mid indices of A and B
  → If mid element of A is greater than mid element of B, we can safely ignore second half of A; else we can ignore second half of B.

* Then the worst case is $\in O(\log m + \log n)$ $m \to$ A's length
$n \to$ B's length

3) The Contiguous Subset With The Largest Sum

This algorithm divide the given array in two halves and return the maximum of following three: Maximum subarray sum in left half by using recursive call. Maximum subarray sum in right half by using recursive call. Then, maximum subarray sum such that the subarray crosses the midpoint. find-max-crossing-subarray function is written to calculate third. This is not recursive function. It can easily find the crossing sum in linear time. It is basic, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid+1 and ending with sum point on right of mid+1. Finally, combine the two and return the result. I used two global variables to find the subarray's first and last indexes. find_max-subarray is a recursive method and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \Theta(n)$$

$$\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(2T(n/2) + n/2) + n = 4T(n/4) + n + n = 4T(n/4) + 2n \\
&= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + n + 2n = 8T(n/8) + 3n \\
&= 8(2T(n/16) + n/8) + 3n = 8T(n/16) + n + 3n = 16T(n/16) + 4n \\
&\qquad = 32(n/32) + 5n \\
&\qquad \vdots
\end{aligned}$$

Finally $n * T(1) + \log_2(n) * n \in O(n \log n)$ //

4) BIPARTITE GRAPH - DECREASE AND CONQUER

I am used to DFS algorithm in this question. DFS algorithm is decrease by a constant algorithm. The size of an instance is reduced by the same constant on each iteration of the algorithm.

Bipartite graph as we can divide it into two sets U and V with every edge having one end point in set U and the other in set V. There are two ways to check for Bipartite graph:

1- A graph is bipartite if and only if it is 2-colorable
2- A graph is bipartite if and only if it does not contain an odd cycle.

Now using DFS, (not BFS) I will check if the graph is two-colorable (mark 1 or 0 in code) or not.

The main idea is to assign to each vertex the color that differs from the color (1 or 0) of its parent in the depth-first search tree, assigning colors in a preorder traversal of the depth-first -search tree. If there exists an edge connecting current vertex to a previously-colored vertex with the same color, then we can say that the graph is not bipartite. In the code I'm sending a subarray of -1 as many vertex numbers to check for it.

The worst case running time is $O(n+m)$ where n is number of vertices and m is number of edges in the graph.

Note: $O(m)$ may vary between $O(1)$ and $O(n^2)$, depending on how dense the graph is. Ex, in adjacency matrix, complexity is $O(n^2)$.

5) FIND BEST DAY - DIVIDE AND CONQUER

The find-good-day-index function returns the index of the largest element in the array. It does this by using the divide-and-conquer method. It takes an array and its length as a parameter. There are two base conditions. These conditions are one or two in length. Then the problem is minimized by dividing it into two subarray. This is where the recursive part comes in. Dividing is done in the middle. Then the index of the larger one is returned. find-good-day takes coin and price array as parameters. The lengths of these arrays must be equal. If not, the error message is displayed. Then the difference is created by taking the differences and find-good-day-index function is called. 1 must be added to the returned result to find the correct result.

The find-good-day-index function's worst-case complexity is $2T(n/2)+1$. The find-good-day functions for loop is $O(n)$ and total worst-case is:

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

We can use Master Theorem in this question.

$T(n) = a\,T(\frac{n}{b}) + f(n)$ $\quad\quad f(n) = n^d \quad (d \geq 0)$

$a=2 \quad b=2 \quad d=1 \quad\quad b^d = 2 \quad\quad a = b^d \quad$ Case 2

So, worst-case complexity is $\in O(n \log n)$.