PART 1

Name / Surname : Yunus Emre Geyik
No : 1801042635

CSE 321 - HW3

Question 1

(a) Recurrence relations of the following algorithm is $\boxed{tmp = alg1(L[0 \dots n-2])}$

Algorithm alg1(L[0 … n-1])
    if (n == 1) return L[0] } $O(1)$
    else
        tmp = alg1(L[0 … n-2]) } $T(n-1)$
        if (tmp <= L[n-1]) return tmp } $O(1)$
        else return L[n-1]

$$\boxed{T(n) = T(n-1) + O(1)}$$

$$T(n) = \underbrace{T(n-1)} + 1 \quad , n > 0 \quad , \overbrace{T(1) = 1}^{\text{when } n=1}$$

$$T(n-1) = T(n-2) + 1$$

$$T(n-2) = T(n-3) + 1$$

$$T(n) = (\underbrace{T(n-2)} + 1) + 1$$

$$T(n) = (T(n-3) + 1) + 2$$

$$= T(n-3) + 3$$

k times

$$T(n) = T(n-k) + k$$

Assume that $n = k + 1$

$$T(n) = T[n - (n-1)] + (n-1)$$

$$T(n) = T(1) + (n-1) \Longrightarrow \boxed{T(n) = n}$$

$$\boxed{T(n) \in O(n)}$$

Question 1

(b) Reccurrence relations of the following algorithm

tmp1 = alg2 (X[1 ... flr])
tmp2 = alg2 (X[flr+1 ... r])

```
Algorithm alg2 ( X[l ... r])
    if (l==r) return X[l]                    } O(1)
    else
        flr = floor ((l+r)/2)                } O(1)
        tmp1 = alg2 ( X[l ... flr])          } T(n/2)
        tmp2 = alg2 ( X[flr+1 ... r])        } T(n/2)
        if (tmp1 <= temp2) return tmp1       }
        else return tmp2                     } O(1)
```

$$\boxed{T(n) = 2\,T(n/2) + O(1)}$$

Using via Master Theorem

$a = 2, \quad b = 2, \quad f(n) = 1 \rightarrow f(n) \in O(n^{\log_b a})$

$n^{\log_b a} = n^{\log_2 2} = n$

then,

$T(n) = \Theta(n^{\log_b a})$

$$\boxed{T(n) = \Theta(n)}$$

Conculusion

After calculating the time complexity of the two algorithms, I suggest using the first algorithm to solve the same problem. Becouse while the first algorithm works in $O(n)$ time, the other algorithm works in $\Theta(n)$ time. Means that, the second algorithm definitely runs in n time. However, the first algorithm works at most n times, so it can work in less than n times, considering this possibility, it would be cliver to use the first algorithm.

PART 2

Question 2

The algorithm finds the values of the degrees of x one by one, starting from the highest degree of the polynomial expression to the lowest degree, and multiplies it by the coefficient of that degree, does this for all degrees of x and adds them all, and the value of the polynomial at point x_0 is reached

```
def polynomial ( Coefficients, x):
    sum_value = 0                          } Θ(1)
    n = len ( Coefficients ) -1            }
    for i in range (n, -1, -1):   → O(n)
        power = 1                 } Θ(1)
        for j in range(i):                 }
            power = power * x } Θ(n)       } O(n)
        sum_value = sum_value + (power * Coefficient[i]) } Θ(1)
    return sum   } Θ(1)
```

Time complexity of Brute-Force Polynomial Algorithm is $O(n^2)$

$$\prod_{i=n}^{0} \sum_{j=0}^{i} 1 \Rightarrow O(n^2)$$

Question -2

Discussion Another Algorithm

If we did not use the brute-force algorithm to solve the desired problem and does evaluate the polynomial from the lowest to the highest degree, we would get time complexity $O(n)$.

```
def polynomial ( Coefficient, x)
    sum_value = Coefficient [0]   } Θ(1)
    power = 1
    for i in range (1, n):                          O(n)
        power = power * x                           } Θ(1)
        sum_value = sum_value + (power * Coefficient [i])
    return sum_value   } Θ(1)
```

Time Complexity = $O(n)$

$$\sum_{i=1}^{n} 1 = n \in \Theta(n)$$

PART 3

Question 3

The Algorithm Checks the given string from beginning to end, based on brute-force. If it matches the first given letter value while checking, a check is made again until the second given letter value is found from that point, and when the second given value is found, the counter is increased by one. In this way, the desired result is obtained.

```
def find_counter(first, last, str):
    n = len(str)            }
    counter = 0             } Θ(1)
    counter2=0              }
    for i in str:    -->  O(n)
        if (i == 'x'):
            for j in range(counter, n):          }
                if (str[j] == 'z'):       }      } O(n)
                    counter2 = counter2 +1 } Θ(1) }
        counter = counter +1  ->  Θ(1)
    return counter2  ->  Θ(1)
```

Time complexity of Algorithm = $O(n^2)$

$$\sum_{i=0}^{n} \sum_{j=counter}^{n} 1 \Rightarrow O(n^2)$$

PART 4

Question 4

For simplicity we consider the 2-D case, 2 points $p_i(x_i, y_i)$, $p_j(x_j, y_j)$

Distance between 2 points p_i and $p_j = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

BF Algorithm for CP

- Compute the distance between each pair of points $\binom{n}{2}$ pair
- Find the pair with the smallest distance

Pseudocode for BF - CP

$$
\begin{array}{l}
d \leftarrow \infty \\
\text{for } i \leftarrow 1 \text{ to } n-1 \text{ do} \\
\quad \text{for } j = i+1 \text{ to } n \text{ do} \\
\quad\quad d \leftarrow \min\Big(d, \underbrace{sqrt\big[(x_i - x_j)^2 + (y_i - y_j)^2\big]}_{\text{Basic Operation}}\Big) \\
\quad \text{end for} \\
\text{end for} \\
\text{return } d
\end{array}
$$

⊛ Computation of sqrt is costly. it can be avoided. Only compare $(x_i - x_j)^2 + (y_i - y_j)^2$ since sqrt is strictly increasing

⊛ So, basic operation will be squaring a number

$$CP(n) = \sum_{1}^{n-1} \sum_{i+1}^{n} 1 \Rightarrow \sum_{1}^{n-1} n - i \Rightarrow (n-1) + (n-2) + \dots + 1$$

$$\Rightarrow \frac{n.n-1}{2} \in \Theta(n^2)$$

PART 5

Question 5

(a) The algorithm starts from all indices in the array and calculates and compares the varying sums to find the largest value. The outer loop chooses the starting index, the inner loop starts from the index selected by the outer loop and finds the maximum possible sum by sequentially including the values in the remaining indexes of the array, and compares this maximum with the overall maximum.

```
def most_profit_BF(arr, arr2):
    Max_sum = 0          # Θ(1)
    n = len(arr)         # Θ(1)
    first = 0            # Θ(1)
    Second = 0           # Θ(1)
    for i in range(n):                  # O(n)
        sum = 0                         # Θ(1)
        for j in range(i,n):            # O(n)
            sum = sum + arr[j]          # Θ(1)
            if (sum > Max_sum):         # Θ(1)
                Max_sum = sum           # Θ(1)
                first = i               # Θ(1)
                Second = j              # Θ(1)
    print(arr2[first:second])           # Θ(1)
```

Time Complexity

$O(n^2)$

$$\sum_{i=0}^{n} \sum_{j=i}^{n} 1 \Rightarrow \sum_{i=0}^{n} (n-i)$$

$$\Rightarrow (n-1)+(n-2)+\dots+0$$

$$\Rightarrow \frac{n.n-1}{2}$$

$$\Rightarrow \in O(n^2)$$

arr=[3,-5,2,11,-8,9,-5]

arr2=['A','B','C','D','E','F','G']

Question 5

(b) The algorithm is divided into two parts and the maximum total values in the divided parts are found, then with an auxiliary function, the array is divided from the middle point and again two subarray are obtained, their maximum sums are found and these two results are added. The maximum value of the three possibilities obtained is returned to the main function (findMaxProfit). In the main function, three possibilities occur, and in these three possibilities, the maximum value is returned and the maximum profit is reached.

Time Complexity: $\boxed{T(n) = 2T(n/2) + O(n)}$

Using with Master Theorem:

$a = 2$, $b = 2$, $f(n) = n$

$n^{\log_b a} = n^{\log_2 2} = n$ $\longrightarrow$ $f(n) \in \Theta(n^{\log_b a})$

$f(n) \in \Theta(n)$

So,

$T(n) = \Theta(n^{\log_b a} . \log n)$

$\boxed{T(n) = \Theta(n . \log n)}$ //

```
import sys
def findMaxProfit(arr, low, high):
    if (low == high)
        return arr[low]
    middle = (low + high) // 2
    leftSum = findMaxProfit(arr, low, middle)
    rightSum = findMaxProfit(arr, middle+1, high)
    MergeSum = mergeSubArrays(arr, low, middle, high)
    maxProfit = max(leftSum, rightSum, MergeSum)
    return maxProfit

def mergeSubArrays(arr, low, middle, high):
    sum = 0
    leftSum = -sys.maxint
    for i in range(middle, low-1, -1):
        sum = sum + arr[i]
        if (sum > leftSum):
            leftSum = sum
    sum = 0
    rightSum = -sys.maxint
    for i in range(middle+1, high+1):
        sum = sum + arr[i]
        if (sum > rightSum):
            rightSum = sum
    mergeSum = leftSum + rightSum
    return max(mergeSum, leftSum, rightSum)
```

$\Theta(1)$ (lines `if (low == high)` … `middle = (low + high) // 2`)

$T(n/2)$ (`leftSum = …`), $T(n/2)$ (`rightSum = …`), $O(n)$ (`MergeSum = …`), $\Theta(1)$ (`maxProfit = …`, `return maxProfit`)

$O(n)$ (`mergeSubArrays`)

$\Theta(1)$, $\Theta(1)$, $O(n)$, $\Theta(1)$, $\Theta(1)$, $O(n)$, $\Theta(1)$