CSE 321 HOMEWORK 3

Emre Sezer
1901042640

1) a)
```
Algorithm alg1(L[0..n-1])
    if (n==1) return L[0]
    else
        tmp = alg1(L[0..n-2])
        if (tmp <= L[n-1]) return tmp
        else return L[n-1]
```

Each recursion decreases problem by 1 and continues. It is a decrease and conquer algorithm.

$T(n) = T(n-1) + c \rightarrow$ constant, $\Theta(1)$

Best Case: $B(n) = W(n)$. There is no break condition, it will go n times at each case.

Worst Case:

$T(n) = T(n-1) + c$, $T(1) = c$

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

$= T(n-k) + ck$ Assume $n = k+1$

$T(n) = T(1) + c(n-1) = cn$,

$W(n) = T(n) \in \Theta(cn) = \Theta(n)$

Average Case: There is no break condition. At each case it will check until it reaches 0th element. So, $A(n) = W(n)$

$A(n) \in \Theta(n)$

b)
```
Algorithm alg2 (X[l..r])
    if (l==r) return X[l]
    else
      flr = floor((l+r)/2)
      tmp1 = alg2 (X[l..flr])
      tmp2 = alg2 (X[flr+1..r])
      if (tmp <= tmp2) return tmp1
      else return tmp 2
```

Each recursion divides problem into 2 pieces and terminates both of them. It is a divide and conquer algorithm.

$T(n) = 2T(\frac{n}{2}) + c \rightarrow$ constant, $\Theta(1)$ I will apply Master Theorem.

$a=2,\ b=2,\ f(n) \in \Theta(c) = \Theta(1) \quad d \geq 0$

$f(n) \in O(n^{\log_2 2}) = O(n)$ Case 1

$T(n) \in \Theta(n^{\log_2 2})$; $T(n) \in \Theta(n)$

$B(n) = W(n) = A(n) = T(n) \in \Theta(n)$

Best, worst and average time complexities are same. There is no break condition. This algorithm is similar to the merge sort algorithm.

I would prefer first algorithm. It makes less assignments compared to second algorithm. Both of the algorithm's time complexity is $\Theta(n)$.

2)
```
def polynomial (x0, n, arr) :
    result = 0                      } Θ(1)
    for i in range(0, n) :
        temp = 1                    } Θ(1)
        for j in range (0, size-i-1)
            temp = temp * x0        } Θ(1)      } O(n)      } O(n²)
        result = arr[i] * temp + result } Θ(1)
    return result                   } Θ(1)
```

arr contains constants of the polynomial $(a_1, a_2, \dots a_n)$

Inner loop calculates x^k; (k is a number between 0 and n)

Inside outer loop and right after inner loop: it calculates $a_k * x^k$ and adds to the result. Outer loop will be executed n times and it will return result.

There is no best or worst case for this algorithm. At each case loops will iterate max number of times. There is no break conditions.

$B(n) = W(n) = A(n) = T(n)$, $T(n) \in O(n^2)$

$T(n) \notin \Theta(n^2)$. Because inner loop will not iterate n times at each iteration.

It is possible to design an algorithm that has better time complexity. Horner's algorithm which works with O(n) has better time complexity than my algorithm. Also, if i could calculate x^k better than O(n). That algorithm also would have better time complexity.

3)
```
def count_str(str a b):
    counter = 0                          } Θ(1)
    for i in range(len(str) - 1):
        if str[i] == a:
            for j in range(i+1, len(str)):
                if str[j] == b:
                    counter += 1         } Θ(1)
    return counter                       } Θ(1)
```

$O(n)$

$O(n^2)$

It will check every possible substring. First loop is for deciding start index, second loop is for deciding last index of the current substring. If first character of the substring is a, then inner loop will execute and will check each substring starts from i'th index. If current substring's last character is b, counter will increase by 1 and current substring is a desired substring. After loops are executed it will return the counter.

Best Case: It will occur if there is no a in the str. Outer loop will terminate n times and return 0.

$B(n) \in \Theta(n)$

Worst Case: It will occur if every element in the str is a. Inner loop will iterate max number of times.

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

Average Case: Since, there are 2 nested loops.

$T(n) \in O(n^2)$, $A(n) = T(n)$

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

4) For any $k \in \mathbb{Z}^+$ distance function's time complexity is $O(n)$.

$$D = \sqrt{(x_k - y_k)^2 + (x_{k-1} - y_{k-1}) + \cdots (x_1 - y_1)^2} \in O(n)$$

Set minDistance to 0. Set tempDistance to 0.

For all pair combinations of the the set with n elements:

Let say u and v are 2 elements of each pair.

tempDistance = D(u, v) → Distance function

if (tempDistance < minDistance) minDistance = tempDistance.

After loops are terminated return minDistance.

There is no break condition. So, at any case it will iterate for all pair combinations.

$B(n) = W(n) = A(n) = T(n)$

$$T(n) = \binom{n}{2} \cdot n = \frac{n!}{(n-2)!\,2!} \cdot n = \frac{n.(n-1)\cancel{(n-2)!}}{\cancel{(n-2)!}\,2!} \cdot n$$

$$= \frac{n^3 - n^2}{2} \in \Theta(n^3)$$

$$T(n) \in \Theta(n^3)$$

Scanned with CamScanner

5) a)

```
def cluster1(values, names):
    maxProfit = 0 } Θ(1)
    result = []
    for i in range(len(values)-1):
        for J in range(i+1, len(values)):
            temp = 0 } Θ(1)
            for k in range(i, J+1):              } O(n)   } O(n²)   } O(n³)
                temp += values[k] } Θ(1)
            if temp > maxProfit:
                maxProfit = temp                 } Θ(1)
                result = names[i:J+1]
```

It will check every possible consecutive subarray. First loop decides the start index of the subarray, second loop decides the last index of the subarray. Third loop will calculate the profit of the current subarray and sets to the temp. If temp is greater than maxProfit, result is equal to current subarray. After all loops are executed, it will return the result.

There is no best or worst case for this algorithm. At each case loops will be executed max number of times. There is no break condition.

$B(n) = W(n) = A(n) = T(n)$, $T(n) \in O(n^3)$

$T(n) \notin \Theta(n^3)$. Because third loop will not iterate n times at each iteration.

b)
```
def maxSum(low, med, high, arr):
    temp = 0    leftMaxSum = 0   rightMaxSum = 0 } Θ(1)
    for i in range(low-1, med):
        temp = temp + arr[med-i-1]          } O(n)
        if (temp > leftMaxSum)
            leftMaxSum = temp
    temp = 0
    for i in range(med+1, high+1):
        temp = temp + arr[i]                } O(n)
        if (temp > rightMaxSum):
            rightMaxSum = temp
    return max(leftMaxSum, rightMaxSum, leftMaxSum + rightMaxSum)    } Θ(n)

def cluster(low, high, arr):
    if (low == high):
        return arr[low]
    med = (low + high) // 2   # // is integer division
    return max(cluster(low, med, arr), cluster(med+1, high, arr),
               maxSum(low, med, high, arr))
```

Each recursion divides problem into 2 pieces and executes both of them and at each step it calls maxSum function which works with $\Theta(n)$ time complexity.

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$ (I will apply Master Theorem)

$a=2,\ b=2,\ f(n) \in \Theta(n)\quad d \geq 0$

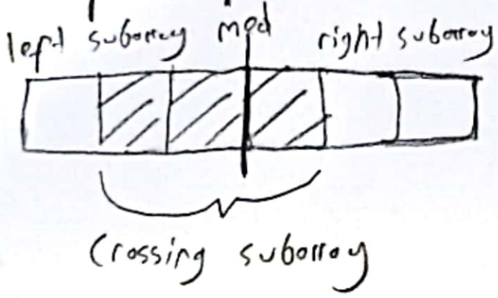
$f(n) \in \Theta(n^{\log_2 2})$ [Case 2]

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

$T(n) \in \Theta(n^{\log_2 2} . \log n)$

$T(n) \in \Theta(n . \log n)$

maxSum function calculates subarray's left side's maximum summation, subarray's righ side's maximum summation and summation of left and righ side's maximum summations. It return's the greatest one among the 3 calculations. With that function at each recursion step it finds the crossing subarray with maximum summation.



It calculates left and right subarray's max summations. It also decides if left subarray or right subarray or crossing sub array is the greatest.

cluster function divides problem into 2 parts at each recursion and executes both of them. At each recursion, it decides if left subarray or right subarray or current subarray has the greatest summation and returns the greatest one. It calculates each subarray's greatest summation with maxSum function.