Burak Yıldırım
1901042609

Q1) a) $T(n) = T(n-1) + 1$
$T(n-1) = T(n-2) + 1$
$\vdots$
$T(2) = T(1) + 1$
$T(1) = 1$
} n

$n \cdot 1 = n \Rightarrow T(n) = \Theta(n)$

b) $T(n) = 2T(n/2) + 1$

$a = 2 \quad b = 2 \quad f(n) = 1$

$n^{\log_2 2} = n \quad n > 1 \rightarrow$ case 1 of master theorem

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

Q2) Algorithm computePolynomial(coefficients[0...n-1], x)
power = length(coefficients) - 1 } $\Theta(1)$
result = 0 } $\Theta(1)$
$\Theta(n)$ { For i = 0 to length(coefficients) - 1
result += coefficients[i] * power(x, power) } $\Theta(1)$
power -= 1 $\Theta(1)$
return result } $\Theta(1)$

Complexity : $\Theta(n)$

I don't think it's possible to design a better algorithm because no matter what, we have to traverse through all the coefficients to compute the polynomial.

Q3) Algorithm count_substrings(text, start, end)

```
results = [] } Θ(1)
for i = 0 to length(text) - 1                                   } Θ(n²)
    indexes = [] } Θ(1)
    if(text[i] == start && i+1 < length(text)) } Θ(1)
        for j = i+1 to length(text) - 1                         } Θ(n²)
            if(text[j] == end) } Θ(1)                           } Θ(n)
                indexes.push(j) } Θ(1)
    for j = 0 to length(indexes) - 1                            } Θ(n)
        results.push(text[i... indexes[j]]) } Θ(1)
return length(results) } Θ(1)
```

Complexity: $\Theta(n^2)$

Q4) Algorithm find_closest(set[0...n-1][0...m-1])

```
closest_distance = infinity } Θ(1)
for i = 0 to n-1                                                    } Θ(m²·n²)
    if(i+1 < n) } Θ(1)
        for j = i+1 to n-1                                          } Θ(m·n)
            sum = 0 } Θ(1)
            for k = 0 to m-1                                        } Θ(m)
                sum += power((set[i][k] - set[j][k]), 2) } Θ(1)
            sum = sqrt(sum) } Θ(1)
            if(sum < closest_distance) } Θ(1)
                closest_distance = sum } Θ(1)
return closest_distance } Θ(1)
```

Complexity: $\Theta(m \cdot n^2)$

Q5) a) Algorithm find-cluster (branches [0...n-1])

```
cluster = [] } Θ(1)
max-profit = 0 } Θ(1)
for i = 0 to n-1                                          } Θ(n³)
    array = [] } Θ(1)
    for j = i to n-1                                      } Θ(n²)
        array.push(branches[i...j]) } Θ(n) } Θ(n²)
    for j = 0 to length(array)-1                          } Θ(n²)
        temp = [] } Θ(1)
        sum = 0 } Θ(1)
        for k = 0 to length(array[j])-1                   } Θ(n)
            temp.push(array[j][k].name) } Θ(1)
            sum += array[j][k].profit } Θ(1)
        if (sum > max-profit) } Θ(1)
            max-profit = sum } Θ(1)
            cluster = temp } Θ(1)
return cluster } Θ(1)
```

complexity: $\Theta(n^3)$

P.S. branches array is an array of Branch classes which have name and profit attributes.

b) Algorithm find_max_profit(cluster[0..high], low, high)

```
if(low == high)
    return cluster[low]

middle = floor((high+1)/2)
right_s = find_max_profit(cluster, middle+1, high)
left_s = find_max_profit(cluster, low, middle)
sum = 0
temp_left = -infinity
for i = middle to low
    sum += cluster[i]
    if(sum > temp_left)
        temp_left = sum
sum = 0
temp_right = -infinity
for i = middle+1 to high
    sum += cluster[i]
    if(sum > temp_right)
        temp_right = sum
middle_s = maximum((temp_right + temp_left), temp_left,
                   temp_right)
return maximum(middle_s, left_s, right_s)
```

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

$a = 2 \quad b = 2 \quad f(n) = n$

$n^{\log_2 2} = n \qquad n = n \rightarrow$ master theorem case 2

$T(n) = \Theta(n^{\log_2 2} \cdot \log n) = \Theta(n \cdot \log n)$

Complexity : $\Theta(n \cdot \log n)$