CSE321 Homework 5

Ahmet Tuğkan Ayhan ..../......./.......

1901042692

Question 1

```
Algorithm maxProfit ( profitRates [0... n-1] )
    size ← size of the profitRates (which is n)     }
    localMaxProfit ← 0                               } Θ(1)
    globalMaxProfit ← -∞                             }

Θ(n) { for i=0 to size do :
           localMaxProfit = max( profitRates[i] , profitRates[i] + localMaxProfit )  }
           if localMaxProfit > globalMaxProfit                                         } Θ(1)
               globalMaxProfit ← localMaxProfit                                        }

    return globalMaxProfit
```

$$T(n) = \sum_{i=0}^{n} 1 = n+1 \rightarrow T(n) \in \Theta(n)$$

Time Complexity

Space Complexity $\rightarrow \Theta(1)$, because inside the algorithm there is no extra memory allocation related to n

Algorithm

* With this algorithm we use previously calculated localMaxProfit to calculate next index' localMaxProfit. By doing so we improve our algorithm from $\Theta(n^2)$ complexity to $\Theta(n)$. Inside for loop first we calculate localMaxProfit then we compare it with the globalMaxProfit. If global is less than local then we assign local to globalMaxProfit.

Question 2

Algorithm maxObtainableValue(rodLength, prices) $\nearrow n$
    values ← fill with (rodLength+1) number of zeros (int array) → $\Theta(n)$

$\Theta(n^2)$ {
    for i=1 to rodLength+1 do :
        maxValue ← $-\infty$ → $\Theta(1)$
        $\Theta(n)$ { for j=0 to i do :
            maxValue ← max(maxValue, prices[j] + values[i-j-1]) → $\Theta(1)$
        values[i] ← maxValue → $\Theta(1)$ }

    return values[rodLength] → $\Theta(1)$

$$T(n) = \sum_{i=0}^{n}\sum_{j=0}^{i} 1 = \sum_{i=0}^{n} i+1 = \underbrace{(0+1)}_{1} + \underbrace{(1+1)}_{2} + \underbrace{(2+1)}_{3} \ldots + (n+1) = \frac{(n+1).(n+2)}{2}$$

$$\cong n^2$$

* Time Complexity → $T(n) \in \Theta(n^2)$

* Space Complexity → $\Theta(n)$, because we create an array with the size of $n+1$

Algorithm

* Values array stores the maximum value achieved from a rod with length i
* For the outer for loop, before assigning maxValue to values[i] first we must find what is the max obtainable value. Inside to inner for loop this is what we are doing.
* First we divide the rod with length i into two rods with lenghts of j and i-j. Then, since we already calculated (i-j) length rod's maxValue we get it from values[i-j-1]. "-1" used because if we are looking for 2nd rod we get it by values[2-1] → values[1].

* Finally we sum that maxValue with rest of the rod (which is j). we get it's price by doing prices[j]. Then we assign either current maxValue or new sum into that rod's value (which is values[i])

Question 3

Algorithm maxPrice(prices, weights, maxWeight) :

```
    size ← n (which is length of the prices and weights arrays)  }
    price ← 0                                                     } Θ(1)
    weight ← 0                                                    }

    for i = 0 to size do :  → Θ(n)
        if (weight+weights[i] <= maxWeight)                              }
            weight  ← weight + weights[i]                                }
            price   ← price + prices[i]                                  }
        else                                                             } Θ(1)
            remainRatio ← (maxWeight - weight) / weights[i]              }
            weight  ← weight + (remainRatio * weights[i])                }
            price   ← price + (remainRatio * prices[i])                  }
            break                                                        }

    return price
```

Time Complexity

$$* \; T(n) = \sum_{i=0}^{n} 1 = n+1 \simeq n \rightarrow \boxed{T(n) \in \Theta(n)}$$

* Space Complexity = $\Theta(1)$, because there is no allocation that is related to n (increases with n)

Algorithm

* Actually there are 2 cases for this problem. prices and weights are given sorted in decreasing order according to price/weight ratio or not.

* If these arrays are given sorted, the algorithm's time complexity will be $\Theta(n)$
* If these arrays are not sorted, then because of sorting is needed with the best sorting algorithm in terms of time complexity, this algorithm's time complexity will be $\Theta(n \log n)$
* Inside the algorithm, I assumed arrays are given sorted because of that with the best price/weight ratio elements are stored first. Until last element or max weight is reached I sum up weights and prices. After we need to slice cheese I took the ratio and added the last piece to weight and price. Finally I returned the price

Question 4

Algorithm maxCourse(startTimes, finishTimes)

size ← length of the arrays startTimes and finishTimes (Which is n) → $\Theta(1)$

if size < 1
    return 0
$\} \Theta(1)$

lastFinishTimeIndex = 0
count ← 1
$\} \Theta(1)$

$\Theta(n)$
for time = 1 to size do :
    if (startTimes[time] >= finishTimes[lastFinishTimeIndex])
        lastFinishTimeIndex ← time
        count ← count + 1
$\} \Theta(1)$
return count

Time Complexity : $\sum_{i=1}^{n} 1 = n \rightarrow \boxed{T(n) \in \Theta(n)}$

Space Complexity : $\Theta(1)$, because there is no extra memory allocation inside the algorithm that increases with n (array size)

Algorithm

* There are also two cases for this algorithm too like in question 3.
* First case ; finish times can be given sorted in increasing order
  Second case ; finish times can be given unsorted.

* Since finish times inside the homework pdf is sorted like case 1, I assumed we've given a sorted array and didn't sort anything.

* For the first case time complexity is $\Theta(n)$ and for the second case time complexity $\Theta(n \log n)$ because of sorting

* Algorithm itself compares last selected finish time with rest of the courses, If finish time is not after start time then it increments course count and assigns this course's finish time to lastFinishTimeIndex.