CSE 321-Introduction to Algorithm Design
Homework 5

180104203
Ozan GEÇKİN

Q1) Design a dynamic programming algorithm that finds the maximum profit belonging to the most profitable cluster.

```
def cluster(station, n):
    nextMax = station[0]          → O(1)
    currMax = station[0]          → O(1)

    for i in range(1,n):
        currMax = max(station[i], currMax + station[i]) → O(1)
        nextMax = max(nextMax, currMax) → O(1)        } Θ(n)
    return nextMax
```

The problem with this question is a largest subarray sum. I did it in the form of dynamic programming. I set the first elements of my array as base and did the calculation.

We have a for loop that contains operations in constant time, which we are looking at from a time complexity analysis point of view. So Time Complexity is $\Theta(n)$

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

b) Compare the algorithm with the algorithms you previously proposed in terms of complexity

→ My previous application was made with Divide and Conquer algorithm design. It was recursive function and it is Time Complexity was $\Theta(n \log n)$. Therefore, dynamic programming is faster. Because it does not contain recursive functions. It uses the memorization technique.

My previous solution Recursive

```
def maximumProfit(arr, start, last)
    if(start == last):
        return arr(start)
    m = (start + last)/2
    return max(maximumProfit(arr, start, m),
               maximumProfit(arr, m+1, last),
               maxCrossingSum(arr, start, m, last))
Def maxCrossingSum(arr, start, m, last)
    loop i to(m, start-1, -1)   → Σ_{i+1}^{n} 1 = n
    {
    loop i to(m+1, last+1)      → Σ_{i+1}^{n} 1 = n
    }
    return max(left_sum, right_sum, left_sum, right_sum
```

Time Complexity

$T(n) = 2T(\frac{n}{2}) + n(n)$ ↙ $T(n) = 2n = \Theta(n)$

$T(n) = 2T(\frac{n}{2}) + 2n$

Using Master Theorem

$a=2 \quad b=2 \quad c=2n=n$

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

Time Complexity Dynamic

$\Theta(n)$

So, Dynamic programming is faster.

Q2)

180106210 3
Ozan GEÇKİN

```
def candy(maxLength, length, price, n):
    K = [[0 for x in range(maxLength+1)] for x in range(n+1)]
    for i in range(n+1):
        for j in range(maxLength+1):
            if i==0 or j==0:
                K[i][j] = 0
            elif length[i-1] <= j:
                K[i][j] = max(price[i-1] + K[i-1][j-length[i-1]], K[i-1][j])
            else:
                K[i][j] = K[i-1][j]
    return K[n][maxLength]
```

Our problem here is the same as our knapsack problem. Weights, values and maximum movable weight in the knapsack problem. In our problem, values are prices. Capacity, is maximum length. Weight is the total length. We have to choose some numbers so that the numbers are the most profitable choice

Time complexity for this problem is calculated as $\Theta(n \times maxLength)$. The reason behind this, is there are 2 for loops with ranges size and maxlength.

Since I'm doing dynamic programming, I did iterative.

size = n

maxlength = m

$$\sum_{i=1}^{n} \sum_{i=1}^{m} 1 = \sum_{i=1}^{n} m \Rightarrow T(n) = \Theta(n,m)$$

180104010 3
Ozan GEÇKİN

Q3)

```
def choseCheese(pi, wi, W):
    index = [0]*len(pi)
    ratio = [v/w for v,w in zip(pi, wi)]
    index.sort(key=lambda i:ratio[i], reverse=True)  → O(nlogn)
    maxPrice = 0
    cheese = [0]*len(pi)
    for i in index:
        if wi[i] <= W:
            cheese[i] = 1
            maxPrice += pi[i]
            W -= wi[i]
        else:
            cheese[i] = W/wi[i]
            maxPrice += pi[i]*W/wi[i]
            break
    return maxPrice, cheese.
```

O(n)

This problem is a typical Fractional Knapsack problem. We are sending two arrays. In this question, I sent cheese prices and cheese weights. At first I do a sort operation. This sort takes O(nlogn) time. It goes into the for loop after it. This for loop runs in O(n) time. Basic operations run in O(1) time.

The result is the time complexity O(nlogn) that the entire function has, But O(n) also works if the array is in reverse order.

I did it with the Greedy Approach. I calculated the price/weight ratio. I listed this ratio. I listed this ratio. I'm adding the big odds in order. I add until capacity is full. Even if it doesn't fit fully in the capacity, I put a certain part of it.

Q4)

150104210?
Ozan GÜCÜKLÜ

```
def maximumCourseSelection(lesson, start, finish, n):
    A = [0]*n
    A[0] = lesson[0]
    k = 0
    iter = 0
    for i in range(1,n):
      if start[i] >= finish[k]:
        iter = iter+1
        A[iter] = lesson[i]
        k = i
    return A
```

$\Theta(n)$

The problem here is a typical Activity Selection problem. To select an activity that starts at a certain and ends at a certain time without conflict. For this to be greedy, we are selecting the first element of the courses as our baseline.
We need to follow the last activity to see which activity will start after the selected activity is finished I'm doing it with k for this.
Finally I return array A. There is a for loop containing basic statements that run at constant time. As result Time Complexity $\Theta(n)$

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