HW5 REPORT

Kahraman Arda KILIÇ – 1901042648

Q1.

In this question, we are asked to find the maximum profit of a cluster that contains adjacent branches. To solve this problem by dynamic programming, we need to hold the latest maximum profit in an array. By using this method, we will calculate the maximum profit with 1 loop. But we will use O(n) space. F[i-1] indicates the latest maximum profit before the arr[i]. At the end, we will have the maximum profit at the nth index of F.

```
def max_profit(arr,n):
    F = [0] * (n+1)
    F[0] = arr[0]
    print(arr)

    for i in range(1,n):
        F[i] = max(arr[i]+F[i-1] , arr[i])
        F[n] = max(F[n], F[i])
        print(arr[i], F[i-1], F[i])

    print(F)
    return F[n]
```

a)

In this question there is only 1 loop:

⇨ $\sum_{i=1}^{n-1} 1 = 1 + 1 + \cdots + 1 = (n-1) \cdot 1 = n - 1 = o(n)$

b) It was O(n * logn) in the 3rd homework. The growth rate of n* logn > n, so for the complexity, O(n) > O(nlogn) (In terms of efficiency). Dynamic programming has better performance.

Q2.

I hold the maximum value in the arr. It's nth index holds the maximum value. For each iteration in the outside loop, the arr gets a new value. For each iteration in the inner loop, the maximum value is calculated.

```
def candy(prices, n):
    arr = [0 for x in range(n+1)]
    arr[0] = 0

    for i in range(1, n+1):
        highest = math.inf * -1
        for j in range(i):
            highest = max(highest, prices[j] + arr[i-j-1])
        arr[i] = highest

    return arr[n]
```

There are 2 loops.

⇨ $\sum_{i=1}^{n} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n} 1 + 1 + 1 \ldots + 1 = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots n = \frac{n \cdot (n+1)}{2} = 0(n^2)$

Q3.

This question is similar to the Fractional Knapsack. The main idea is we need to sort the array by looking the value / weight rate. After that, until we do not have remaining weight, we will get an element (starting from the first) from the array and decrease the weight and increase the totalPrice. If we do not have enough space for a piece, then we need to find the fractional to put that cheese into the box. After this operation, the function returns the totalPrice.

```
def keyFunc(product):
    return product[1] // product[0]


def cheese(arr, n ,weight):
    totalPrice = 0
    arr.sort(key = keyFunc)
    for i in range(0,n):
        if weight - arr[i][0] >= 0:
            weight -= arr[i][0]
            totalPrice += arr[i][1]
        else:
            totalPrice += arr[i][1] * (weight / arr[i][0])
            weight -= arr[i][0] * (weight / arr[i][0])
            break

    return totalPrice
```

There is 1 loop and sort operation (which has O(nlogn) complexity).

- ⇨ $\sum_{i=0}^{n-1} 1 = 1 + 1 + 1 + \cdots 1 = n \cdot 1 = 0(n)$ is the complexity of the loop
- ⇨ $Sort\ operation\ has\ bigger\ growth\ rate$. So the complexity of this algorithm is O(nlogn)

Q4.

This question kind of similar to the Question 3. But we do not have a weight here. We just need to find the maximum number of courses for a student. Firstly, we need to consider the finish hours for sorting. Because student cannot enroll 2 courses at the same time. We will start with the first course that is finished earlier than others. So that is why, first element of the attended array is first element of the sorted array. For every iteration in the loop, we compare the start time of a course with the finish time of the latest enrolled course. If the start time is higher or equal then student can enroll for this course. At the end of the function, we return the array which contains the courses student attended.

```python
def sortKey(obj):
    return obj[2]


def max_course(courses,n):
    attended = []

    courses.sort(key = sortKey)

    num_attended = 0
    attended.append(courses[0])

    for i in range(1,n):
        if courses[i][1] >= courses[num_attended][2]:
            attended.append(courses[i])
            num_attended = i

    return attended
```

There is a loop.

- ⇨ $\sum_{i=1}^{n} 1 = 1 + 1 + 1 + \cdots 1 = n \cdot 1 = 0(n)$
- ⇨ *But we have to consider the sort method as we stated above. It has* $O(nlogn)$ *complexity,*
- ⇨ *So the complexity of this algorithm is* $O(nlogn)$