CSE 321 HW5

Emre SEZER - 1901042640

1) a)

```
def cluster(arr, length)
    max_now = 0                                       } Θ(1)
    max_overall = -inf #negative infinity             }
    for i in range(0, length):                        }
        max_now = max_now + arr[i]            }       }
        if (max_overall < max_now):           }       }
            max_overall = max_now             } Θ(1)  } Θ(n)
        if (max_now < 0):                     }       }
            max_now = 0                       }       }
    return max_overall } Θ(1)
```

arr is the array with profit values. length is the length of the arr. max_now is for evaluating the current consecutive sub-array. If max_now > max_overall, For i'th index the most profit possible is max_now. That repeats length times. But if max_now < 0 at any index, that means current sub-array is not profitable and resets max_now to 0. With that algorithm i find the most profit.

$$T(n) = \Theta(1) + \Theta(n) + \Theta(1) = \Theta(n)$$

There is no break condition. So, it will iterate n times every time. Because of that, most proper time complexity is $\Theta(n)$, not $O(n)$.

b) My previous algorithm's time complexity is $\Theta(n.\log n)$.

Current algorithm's time complexity is $\Theta(n)$.

for $\forall n \in \mathbb{Z}^+ \quad n < n.\log n$

In terms of time complexity current algorithm's time complexity is better than previous one.

Recurrence Relation :

$$cluster(i) = max(cluster(i-1) + arr[i], arr[i])$$

We are trying to find the most profitable sub-array. So, at each step it checks if continue with current sub-array and add the current element to it, or set a new sub-array with current element. It decides the one with the greater value.

2)
```
def candy(price_arr, length):
    temp_arr = [] # empty array } Θ(1)
    for i in range(length+1):       }
        temp_arr.append(0)          } Θ(n)
    for i in range(1, length+1):
        max_price = -inf # negative infinity } Θ(1)
        for J in range(i):
            if(price_arr[J] + temp_arr[i-J-1] > max_price):   } Θ(n)
                max_price = price_arr[J] + temp_arr[i-J-1]
        temp_arr[i] = max_price                                } Θ(n^2)
    return temp_arr[length] } Θ(1)
```

---

price_arr is the array with price values. length is the length of the price_arr. Initially sets temp_arr to a zero array with length+1 elements. It calculates max profit for each i and stores it at the temp_arr[i]. For that: it iterates J from 0 to i.

If (price_arr[J] + temp_arr[i-J-1] > max_price) max_price =

With that approach max profit is stored in the temp_arr[length]. Since loop will iterate length times and each index's max_profit is stored in the temp_arr[i].

$$T(n) = \Theta(1) + \Theta(n) + \Theta(n^2) + \Theta(1) = \Theta(n^2)$$

There is no break conditions. So, loops will iterate until the end. That means $A(n) = W(n) = B(n) = T(n)$

$$= \Theta(n^2)$$

Recurrence Relation :

$$c_n = \max \{ p_i + c_{n-i} \; ; \; 1 \leq i \leq n \}$$

At each i we calculate the most profitable c_n by calculating $(p_i + c_{n-i})$ n-times. It continuously checks previous c's and finds the most profitable c_n.

3)

```
class Cheese:
    def __init__(self, price, weight):
        self.price = price
        self.weight = weight
        self.ratio = 0


def sort_ratio(e):
    return e.ratio


def cheese(arr, Weight):
    result = 0
    ratios = []                                           Θ(1)
    for i in range(len(arr)):
        arr[i].ratio = (arr[i].price / arr[i].weight)     Θ(n)
        ratios.append(0)
    arr.sort(reverse = True, key = sort_ratio)            Θ(n logn)
    weight = 0                                            Θ(1)
    for i in range(len(arr)):
        if(weight + arr[i].weight < Weight):
            weight = weight + arr[i].weight               O(n)
            ratios[i] = 1
        else:
            ratios[i] = (Weight - weight) / arr[i].weight
            break
    for i in range(len(arr)):
        result = result + arr[i].price * ratios[i]        Θ(n)

    return result                                         Θ(1)
```

3) I created a cheese class first. This class has integers price, weight and ratio. First, it calculates price and weight ratios and stores them. Later on, it sorts the array according to ratio's. It checks, if i'th element in the array doesn't exceed the max capacity multiplies it's price with 1 and adds it to the result. Else, multiplies i'th element's price with (Max capacity - Rest of the capacity) / i'th element's weight and adds it to the result. Breaks the loop. Returns the result.

$$T(n) = \Theta(1) + \Theta(n) + \Theta(n\log n) + O(n) + \Theta(n) + \Theta(1)$$

$$= \Theta(n\log n)$$

4)

```
class Course:
   def __init__(self, start_time, finish_time):
      self.start_time = start_time
      self.finish_time = finish_time


def sort_finish_time(e):
   return e.finish_time


def courses(arr):
   arr.sort(key = sort_finish_time)   ) Θ(n logn)
   last_time = arr[0].finish_time     ) Θ(1)
   result = 1

   for i in range(1, len(arr)):
      if(arr[i].start_time >= last_time):
         result = result + 1            ) Θ(n)
         last_time = arr[i].finish_time

   return result   ) Θ(1)
```

4) I created a Course class. It has start_time and finish_time integers. arr is an array with course variables inside it.
It sorts arr according to finish_time values. Since, arr is sorted sets last_time to arr[0]. arr[0] is the course with smallest finish_time. Then, inside a for loop (0 to len(arr))
it checks if arr[i].start_time $\geq$ last_time. If it is increases result by 1 which was 1 initially. After checking every element returns the result.

Sorting takes $\Theta(n\log n)$ time.

$$T(n) = \Theta(n\log n) + \Theta(1) + \Theta(n) + \Theta(1)$$

$$= \Theta(n\log n)$$