Hannah Kwon
705182275
Professor Hsieh

Homework #4

Computer Science 180 - Introduction to Algorithms and Complexity

Question 1

1)

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Total
Data	15	15	15	15	15	15	-
Reboot 0	8	4	2	1	0.5	0.25	15.75
Reboot 1	8	4	-	8	4	2	24
Reboot 2	8	-	8	-	8	4	**28**

Optimal solution for this example would be rebooting twice on day 2 and day 4 that gets a total max of 28. It is bigger than rebooting once on day 3 which gets a total of 24.

2) This question can be solved by using the dynamic programming algorithm which stores the optimal solution in a newly made array to store information. Let us call this new array DP. In every index of the DP array (DP[i]), we will be storing the highest possible data processed until that index of day i + 1 (since day starts with 1 and index starts with 0).
The value of DP[i] will be the maximum of two cases :
Case 1)If we reboot the day before, the computer is able to process the maximum amount of data which is s[0], in this example 8. Then, we need to take the minimum between this value and the current available data, in this example 15. Then, we need to sum with the D[i-2] value which is the last value before the reboot.
Case 2) However, if we don't reboot the day before, we need to know how much data the computer is able to process which depends on the day of the last reboot. Then, we need to take the minimum between this value and the current available data as we did in case 1. Then, we sum with the D[i-1] value.
This algorithm runs in O(n) time complexity since we only loop through the days once in order to fill the DP array. Also, the space complexity would be O(n) since we are making an additional array to put in the optimal information for every index.

```
x = array of amounts of available data for each day
s = array containing the profile of our system that sparts from a freshly
    rebooted system on day 1
n = array size of x = array size of s
DP = we will be storing the optimization in each index, an array of size n
count = 0 #counts how many days since rebooting

#initialize the first two index of the DP array
#Since the best optimization until day 2 is having no reboots
DP[0] = max(x[0], s[0])
DP[1] = max(x[1], s[1])
#from day3 we have to choose either we should have reboot on the prev days
for i in range (2, n) :
    noRebootPrevDay = DP[i-1] + min(x[i], s[count])
    rebootPrevDay = DP[i-2] + min(x[i], s[0])

    if noRebootPrevDay > rebootPrevDay :
        DP[i] = noRebootPrevDay
        count += 1
    else :
        DP[i] = rebootPrevDay
        count = 0

return DP[-1] #return the last element of the DP array
```

Question 2

For the string to be a palindrome two pointers will be used : one starting from the beginning of the string index and the other that is starting from the end of the string index. Let us say the s and e for each. While we are iterating the string using the s and e pointers, we can encounter two scenarios.

1) The character at s index and e index matches
2) The character at s index and e index doesn't matches

For the 1) case we simply have to increase s pointer by 1 index and decrease e pointer by 1 index.
For the second scenario we have two options :

1) Insert one character at s index to match the character at e index
2) Insert one character at e index to match the character at x index

We won't be actually adding the characters into the string but just calculating the numbers. In case 1, we would be decreasing the e pointer by 1, and s index stays where it is to check if the next characters are matching. In case 2, vice versa. Both the cases add a cost of 1 since we are inserting a letter.

We can then use these two different pairs of new s and e values (s +1, e and s, e -1) to repeat the process and use the minimum result of these as our result for the current pair of s, e. We can see that this can be done by using a recursive function with caching to store the repeated values.

To build a bottom-up iteration, we need to iterate all the combinations of (s, e) where s < e.

```
n = length of the string
# make a matrix of
dp = [[0] * n for _ in range (n)]

for e in range (n) :
    for s in range (e-1, -1, -1) : #iterating the string from the back
          dp[s][e] = dp[s+1][j-1] if s[i] == s[j] else min(dp[i+1][j],dp[i][j-1])+1
return dp[0][n-1]
```

Question 3

To cut this rectangular and return the minimum wastage area, there are two options to cut the rectangular : 1) horizontally 2) vertically. Let's define DP(w, l) to be the wastage of a rectangle with width w and length l (1 <= w <= W, 1 <= l <= L).

1. Horizontal : For a horizontal cut at point x where 1 <= x <= w-1
2. Vertical : For a vertical cut at point y where 1 <= y <= l-1

To get the minimum waste for horizontal cut at point (w, l), a minimum waste until the horizontal cut and the minimum waste after the horizontal cut should be added to get the minimum waste at w, l. This can be written as DP(w, l) = DP(x, l) + DP(w-x, l). For the vertical cut would be vice versa from the horizontal cut. This can be written as DP(w,l) = DP(w, y) + DP(w, l - y).

And when we can decide to make a cut, we will try to get the optimization for both the horizontal and the vertical cuts, which can be presented by

DP(w, l) = min(DP(x, l) + DP(w-x,l), DP(w, y) + DP(w, l - y))

```
minWaste = will be an array of size W * L

#initialize values with the area size
minWaste[w,l] = if (w,l) matches some (ai,bi) then set to 0
              = else set to w * l

#After initializing the matrix, we will be storing the minimum
#possible rectangle and find its minimum wastage area

for w from 1 to W :
    for l from 1 to L:
          if minWaste[w][l] != 0 :
                minHorizontal = inf('float')
                minVertical = inf('float')
                #find minimum waste for horizontal cuts
                for x from 1 to w-1 :
                    if DP(x, l) + DP(w-x, l) < minHorizontal :
                          minHorizontal = DP(x, l) + DP(w-x, l)
                #find minimum waste for vertical cuts
                for y from 1 to l-1 :
                    if DP(w, y) + DP(w, l - y) < minVertical :
                          minVertical = DP(w, y) + DP(w, l - y)
                #compute the minimum of to update matrix values
                minWaste[w][l] = min(minWaste[w][l], minHorizontal, minVertical)

return minWaste[w][l] #return the last element of the matrix
```

The algorithm is first initialized to either 0 or the rectangle area size. Now for each rectangle we calculate the minimum wastage area computed in three different ways. One could be either having the minimum waste when it is horizontally cut, another would be when it is vertically cut, and the other would be the value that was already stored. The value in that index will be replaced with the minimum value of these three options. The time complexity of this algorithm is O(k + WL) and the time required to calculate the minimum value for each possible rectangle would be an O(WL(W+L)) time complexity. If we combine these two time complexity together, this algorithm would be an O(k + WL(W+L)) time complexity algorithm.

Question 4

Question 4

To prove that the vertex cover problem $\leq_p$ the hitting set problem, which means the hitting set problem becomes NP-complete. Let us prove first that the hitting set problem is in NP. This can be shown by exhibiting a set hitting set H which can be done in polynomial time whether H is of size K and intersects each of the sets $B_1, \dots B_m$. This is proving that the hitting set-problem is NP.

Now let us go back to the problem : vertex cover problem $\leq_p$ the hitting set problem. We need to reduce hitting set to a version of vertex cover to connect them in polynomial time. The set A is the set of vertices V in considering an instance of the vertex cover problem in the case of $G=(V, E)$. For every edge $e \in E$, we have a set S_e consisting of the two end-points of e. Then we can see that a set of vertices S is a vertex cover of G if and only if the corresponding elements form a hitting set in the hitting instance.

This can be done in polynomial time since $A=V$ takes $O(n)$ and for each edge $(u,v) \in E$ adding a subset $B_i = u, v$ takes $O(m)$ time.