Chetna Ajay Nainani
Rachana Vimal Gandhi
Harsh Pradeep Nahar

Homework - 2

6.A) Q1) Given:
Coins of unlimited supply ($x_1, x_2 \dots x_n$)
Input $x_1, x_2 \dots x_n$
Value to be totalled = V

Is it possible to make changes for V using coins as denominations $x_1, x_2, x_3 \dots x_n$?

Algorithm:→ (indexing from 1 not 0)

```
coinChange (coin, v)        // method coin change

DP[v+1]        // declaring dynamic programming
                  array with length greater
                  than the value

DP[1] = 0      // for first value v we
                  need 0 coins

DP[2 ... ∞] = ∞   // Setting all the rest
                     elements to max value

length = length (coin)

for i = 2 to v+1
    for j=1 to length
        if (coin[j] <= i)
        rest = DP[i - coin[j])
        if (rest != ∞ AND rest+1 < DP[i])
            DP[i] = rest+1
if DP[v] = ∞
        return -1
else return DP[v]
```

Explanation:→

The final value will be contained in DP, the primary principle is to verify for every coin and value. DP records the amount of coins needed to generate that specific value for i, say a certain i is larger than a specific coin, therefore we may create variations for that value i, because we wish to make v, the ultimate solution will be at i = v at coin[v].

Run-time analysis:-

v // Setting all elements to ∞
v // for the first for loop
length // for second for loop
// Total 4 lines inside the loop

So overall,

$$\text{Runtime} = v \times 4 \times v \times \text{length}$$
$$= v \times 4 \times v \times n$$
$$= v + 4nv$$
$$= O(nv)$$

∴ Runtime = $O(nv)$

6.19) Q2) KCoins ()

total[0] = 0

for i = 1 ... to value
    T[i] = ∞
    for z = 1 ... 6 (coin)
        check if (coin[z] ≤ i & total[i] > 1 + total[i - coin[z]])
            total[i] = 1 + total[i - coin[z]]

check if total(value) > k
        return false
else
        return true

Explanation: →

→ Here we are checking if # of limited coins used to make change will be ≤ k or not.

→ If it is possible to make using i coins then we return true

→ else we return false

→ Total keeps the track of min no. of coins used to make value

Running time → $O(nkv)$

$O(nv)$ proved in last problem, here we consider k iterations inside.

6.7

Q3) We will create a 2 dimensional table T, also, $T[i,j]$ will store the longest palindrome in the string $w_i, \dots w_j$. If symbols w_i and w_j are same, we can assume that they are part of the longest palindrome.

$\therefore T[i,j] = 2 + T[i+1, j-1]$.

If the two symbols are different, then both will not be a part of any palindromic sequence. Hence, $T[i,j]$ will be max of $T[i+1, j]$ and $T[i, j-1]$.

The best case happens when $i = j$ so the answer in this case is 1.

Algorithm:

```
for i = 1 to n                       // Declare all diagonal
    T[i,i] = 1 ;                        cells of matrix as 1
for i = n-1 down to 1                // for each cell
    for j = i+1 to n                    above diagonal
        if (wi = wj)
            T[i,j] = 2 + T[i+1, j-1]
        else
            T[i,j] = max (T[i+1, j], T[i, j-1])
```

Algorithm analysis

The first for loop costs n time. It takes n time to traverse from n-1 down to 1 in the outer for loop and then an additional n time to traverse from i+1 to n, k will be constant time to check if else conditions.

Thus time complexity will be

$$n(n+k)+n$$
$$= n^2 + nk + n$$
$$= n^2$$

$\therefore$ Runtime $= O(n^2)$

6.13) Q4) a)

The following is the sequence of cards: 7, 400, 4, 4

When the first player will be greedy, he will choose the first card. (value 7) Then, the second player will choose the card with value 400.
The optimal strategy for the first player is to choose the last card with value 4. Now, the second player can choose either the value 7 or the remaining card with value 4. Now, the first player will choose the card with value 400. In this way, the first player can win.

The time complexity is $O(n^2)$ as the table size is $n \times n$ which is precomputed.

Also, we can consider the following sequence $\{1, 2, 10, 3\}$

first player : 3
Second player : 10
Here, the first player loses the biggest card due to greed
first player : 2
Second player : 1

b) Algorithm:→

```
Optimal Strategy (S)

d[0...n][0...n] → 0

for i in range (0, n):
    for j in range (0, n):
        if i≤j
            d[i][j] = 0
        else
            d[i][j] = max(S[i] + min(d[i-1, j-1]
                      - S[j], d[i-2, j] - S[i+1])
                      S[j] + min(d[i-1, j-1] -
                      S[i], d[i, j-2] - S[j-1]))

return d[n][n]
```

Analysis:→ Let's take the assumption that player 1 attempts to maximize the score and player 2 attempts to minimize it. Hence player 1 will choose the first card or the last card and it is same for player 2. Player 1 should opt for max strategy in order to maximize the score.

We observe that d[i,j] relies on values of d[x,y] where $i \le x \le y \le j$. This algorithm computes these values in order of increasing j. It will take $O(n^2)$ time because there are $O(n^2)$ values to compute, each of which takes constant time. We store a matrix c[i,j] which is the value that player should choose. Using this choice matrix, player 1 can play optimally.

6.11 Q5) LongestCommonSubsequence (String1, String2)
length1 = len (String1)
length2 = len (String2)

dynamicProg [1 ... length1] [1 ---- length2] → 0

```
for index1 in range (1, length1 + 1)
    for index2 in range (1, length2 + 1)
        check if String1 [index1 - 1] == String2 [index2-1]
            dynamicProg [index1] [index2] =
                dynamicProg [index - 1] [index 2 - 1] + 1

        else
            dynamicProg [index1] [index2] = Max(
                dynamicProg [index1], [index2 - 1],
                dynamicProg [index1 - 1] [index2])

return dynamicProg [length1] [length2]
```

Explanation:

→ Initializing the dynamicProg matrix with 0

→ If the text at [index - 1] and [index 2 - 1] matches then we store dynamicProg [index1 -1] [index 2-1] +1

~~then we sto~~ else we take maximum of [index1] [index 2 - 1] & [index1 - 1] [index2]

Running Time = O(mn)