HOME WORK - I

Problem 1 : 15.1.3

Consider a modification of the rod cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with the solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic programming algorithm to solve this modified problem.

```
MEMOIZED-CUT-ROD-AUX (p, n, r)
if r[n] >= 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX
                            (p, n-i, r))
r[n] = q
return q
```

BOTTOM-UP-CUT-ROD (p, n)

```
1. let r[0..n] be a new array
2. r[0] = 0
3. for j = 1 to n
4.     q = -∞
5.     for i = 1 to j
6.         q = max(q, p[i] + r[j-i])
7.     r[j] = q
8. return r[n]
```

Now, when we consider the cost cutting c

We have to replace line 7 in MEMOIZED-CUT-ROD-AUX (p, n, r) with

$$q = \max(q, p[i] + \text{Memoized-cut-rod-Aux}(p, n-i, r, c) - c)$$

And, line 6 in BOTTOM-UP-CUT-ROD with

$$q = \max(q, p[i] + r[j-i] - c)$$

⇒ We can observe that there's no cutting cost for i = 1 & i = n. When there's no cut, no cutting cost is incurred.

Problem 3 15.3-3

Consider the variant of the matrix chain multiplication problem in which goal is to paranthe-size the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications Does this problem exhibit optimal substructure?

Yes, this problem exihibite optimal sub-structure. The concept of optimal substructure is to find a paranthesization which is most optimal and it is not possible to find a less costly way to paranthesize.

Consider that best case parenthesization is just two set of matrix chains. To maximize scalar multiplications, we must find the optimal substructure, in this case, the highest cost, once we reach that there's no costlier way.

Problem 4 15-2

Give a recursive formula, pseudocode for computing the length of the longest palindrome, correctness justification, and the running time of your algorithm. Also give the pseudo code on how to construct the longest palindrome.

Let us consider 'x' as a string which a set of characters. And let us consider $x[i, \dots, j]$ as a substring of string 'x'. We can find a palindrome of minimum length 2 when $x[i] = x[j]$. If it's not same then we begin to check further comparing $x[i+1, \dots, j]$ and $x[i, \dots, j-1]$

Let us consider maximum length of palindrome as $L(i,j)$

So, $$L(i,j) = \begin{cases} L(i+1, j-1) + 2 & x[i] = x[j] \\ \{L(i+1, j) \; L(i, j-1)\} & \text{otherwise} \end{cases}$$

Pseudocode:

```
LONGEST - PALINDROME (x[1, ... n])
// Input: String : L[i,j] is a 2-D array
// output: length of longest palindrome

for i = 1 to n
    L[i,i] = 1                  -> for each character.

for i = 1 to n
    for s = 1 to n-i
    j = s+i
    L[s,j] = PALINDROME-COST (L, x, s, j)
    L[j,s] = PALINDROME-COST (L, x, j, s)
return L[1,n]
```

NOTE:
$L[i,j]$ stores longest palindrome in $x[i, \dots j]$.

Recursion:

```
PALINDROME-COST (L, x, i, j)
// input: Array L from the LONGEST-PALINDROME
// output: cost of L[i,j]

if i = j
    return L[i,j]
else if x[i] = x[j]
    if i+1 < j-1
        return L[i+1, j-1] + 2
    else
        return 2        -> when single character
else
    return max (L[i+1, j], L[i, j-1])
```

Complexity:

→ first for loop → $O(n)$ time

→ Second for loop → $O(n-i)$ → $O(n)$ time

∴ Total running time of algorithm: is $O(n^2)$

Correctness justification:

Initialization: We consider a string that has one or more charcters in it.

Maintainance: It holds true for first iteration and returns the length of palindrome when there is only one character. If it is more than one character, the longest sub palindrome is calculated. Line 4-7 calculates the length of longest palindrome subsequence.

Termination: When the loop terminates, after it's compiled for all the characters in the string. We get the length of the longest sub sequence palindrome and we can tell that the algorithm is correct.

Problem 2: Consider a variation of the longest subsequence (LCS) problem, which we call a string similarity score (SSS). For two strings S1 & S2, each matching character in a sub sequence gives a score of 3, while each alignment character in S1 or S2 that is not matched, gives a penalty score of -1. For example, ACGTC and CATGCTC have an SSS of $4\times3-4=8$, since an LCS has length 4 & there are one and three unmatched characters in S1 & S2 respectively.

Modify the original LCS dynamic programming algorithm to compute the SSS of two strings ~~dynamically~~. directly. That is, your algorithm must not compute the LCS first and then calculate the SSS from the LCS. (4)

Solution:

```
STRING-SIMILARITY-SCORE (x, y)

m = x.length
n = y.length
let score [0....m, 0....n] be new tables
for i = 1 to m
    score [i, 0] = -i
for j = 1 to n
    score [0, j] = -j
for i = 1 to m
    for j = 1 to n
        if x_i == y_i
            score [i, j] = score [i-1, j-1] + 3
        else if score [i-1, j] >= score [i, j-1]
            score [i, j] = score [i-1, j] - 1
        else score [i, j] = score [i, j-1] - 1
return score [m, n]
```

$$\text{score}(i,j) = \begin{cases} -j & \text{if } i = 0 \\ -i & \text{if } j = 0 \\ \text{score}[i-1, j-1] + 3 & \text{if } i, j > 0 \ \& \ x_i = y_j \end{cases}$$