Student Number: 1226909816
Name: Wei Hng Yeo
CSE450 Assignment 3

1) (a) Since the multiplication is divided into 4 subproblems. Each subproblem also have a size of n/2. The action of dividing the problem is also done in constant time. Then as we conquer the subproblem by adding and doing bit shifting, the time complexity is O(n). Thus, the recurrence relation needed will be T(n) = 4 T(n/2) + O(n) = O(n^2)

   (b) As shown in the question that there are 3 sub problems of (a + b)(c + d), ac, bd. The dividinig of problems into sub problems in this case is O(n). As one of the multiplications is substituted with summation and subtraction, which is not as computationally difficult as multiplication. So the recurrence relation is $T(n) = 3T(n/2) + O(n) = O(n \wedge \log_2(3))$.

2) Constructing a longest common subsequence table of the 2 string X and Y.

		0	1	2	3	4	5	6
		null	B	D	C	A	B	A
0	null	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

   Looking at the entries, 4 is the largest and there are 4 of such longest common sub sequence, so the longest common sub sequence are BCAB, BDAB, BCBA.

3) Firstly, we initialize a 2D array of size 5 * 5 and name it arr_2d. Then fill each entry with -1 meaning entries that are not computed yet.

   Using the concept of dynamic programming, we store computation that have already been done before to ensure that we do not repeated work.

   Let's name the list of integer of {13, 5, 89, 3, 34} as matrix_dimension.

   We firstly pass this array into a recursive function stating that we want to start at index = 1 and end at index n – 1 as argument. The function is matrixChainDP(int[] matrix_dimension, int start, int end).

   In this function we return 0 if start == end.

   If the value of current particular of entry arr_2d[start][end] is not equal to -1, then we return the value stored in the arr_2d[start][end] to prevent repeated work.

Else if it is not -1, we will need to compute that particular entry by using a for loop to loop through k times to compute the matrix range of Ai to Ak and Ak to Aj where k is between the range of start and end, in each of the for loop iteration, we do the following below.

arr_2d [i][j] = Min(arr_2d[i][j], matrixChainDP (matrix_dimension, i, k) + matrixChainDP (matrix_dimension, k + 1, j) + (matrix_dimension [i - 1] * matrix_dimension [k] * matrix_dimension [j]))

Then return arr_2d[i][j].

We then obtain the 2D array result to be as follows:
-1, -1, -1, -1, -1
-1, -1, 5785, 1530, 2856
-1, -1, -1, 1335, 1845
-1, -1, -1, -1, 9078
-1, -1, -1, -1, -1

The result we want is the entry at [1][n - 1] or [1][4] which is 2856.

4) (a) The order of running time would be O(n^2). Because we must check through the entire matrix array each time for the two adjacent with the largest remaining dimension. And doing that for the n matrices would result in an order of O(n^2).

(b) Say if we have 3 matrices of dimension that is multiplied: A1 * A2 * A3.
A1 dimension = 4 * 5
A2 dimension = 5 * 5
A3 dimension = 5 * 2

If we do (A1 * A2) * A3 because A1 * A2 is the biggest remaining dimension, the total multiplication required will be 100 because of 4 * 5 * 5. Then we will have a new matrix A'' where A'' has dimension of 4 * 5. Then when we take A'' multiply by A3, we will need another 40 multiplication derived from 4 * 5 * 2. Total multiplication needed is 140.

But if we do A1 * (A2 * A3), the total multiplication required will be 50 for A2 * A3 as the required multiplication is 5 * 5 * 2 = 50 and then we will be left with a matrix say A' of dimension 5 * 2. Then as we multiply A1 and A', we will get 4 * 5 * 2 = 40 multiplication. Total multiplication = 90.

As can be seen from the above example that if we follow the rule of using largest remaining dimension first to do matrix multiplication, we end up with sub-optimal matrix multiplication.