Dynamic Programming - Exercise

1. Given two strings s1 and s2, implement an algorithm to find the longest common subsequence between s1 and s2.
   Function Prototype: char* LCSlength(char *s1, char *s2);

2. Given two strings s1 and s2, implement an algorithm to identify the longest common subsequence between s1 and s2 by maintaining only length information and not using any other indication for the source of the least length.
   Function Prototype: char* LCS(char *s1, char *s2);

3. Given two strings s1 and s2, implement a bottom-up dynamic programming algorithm to identify a longest common subsequence between s1 and s2 and determine its length by maintaining only the recent two rows of the direction table.
   Function Prototype: char* LCS(char *s1, char *s2);

4. Given two strings s1 and s2, implement an algorithm to print all the longest common subsequences between s1 and s2.
   Function Prototype: void allLCS(char *s1, char *s2);

5. Implement an algorithm to print the way in which 'n' matrices to be multiplied in sequence should be parenthesized so that the number of multiplications is the maximum. Also, print the intermediate tables generated in this process.
   Function Prototype: unsigned long long int MCMmaxParanthesize(int D[], int n);

   where, D is an array containing dimensions of the n matrices.

6. Consider the following recurrence relation:

$$C(n,k) = \begin{cases} 0, & n < 0 \; or \; k < 0 \\ 1, & k = 1 \; or \; k = n \\ C(n-1,k-1) + C(n-1,k), & n \; is \; odd \\ C(n-2,k-2) + C(n-2,k), & n \; is \; even \end{cases}$$

   Given n and k, implement a dynamic programming based algorithm to perform the same.
   Function Prototype: unsigned long int compute(int n, int k);

7. Dimensions of an image I is a pair (height of I, width of I). Consider any two images, referred as I_m and I_n, whose dimensions are denoted by (h_m, w_m) and (h_n, w_n) respectively. A bi-image compression technique compresses two images into a single image by embedding I_m on I_n. The compression technique takes $\Theta(w_m * h_n)$ time. Consider an image R, obtained by the compression of Im and In. The dimension of R is (MAX{h_m, h_n}, MAX{w_m, w_n}). For example, if the dimension of I_1 is (200, 500) and that of I_2 is (150, 320), then the dimension of R (obtained by compressing I_1 and I_2, referred as <I_1, I_2>) is (200, 500), and the time taken for compression is around a constant times 500 * 150. Given the dimensions of 'n' images sequenced from 1 to 'n' based on the time at which they were captured, implement an algorithm to print an order for compressing pairs of images (using the aforementioned compression technique) in such a way that the indices of the images in the output is in an increasing order and the time taken is the least of all such possible orderings. For example, a sequence of 3 images I_1, I_2, I_3 should not be ordered as <<I_2, I_1>, I_3>, even if it takes the least time since the indices are not in an increasing order. If the dimensions of I_1, I_2, I_3 are (20, 11), (30, 10) and (15, 50) respectively, the algorithm should return <I_1, <I_2, I_3>>, indicating that I_2 should be compressed with I_3 and then I_1 with the image resulting from the previous compression, the reason being the time taken for this order of compression is less compared to that of the other possible order.
   Function Prototype: void imageOrder(struct pair D[], int n);