Implement problems A and B below in Java and submit by tonight:

A. Write a Java program to implement a recursive version of the function to compute an element of the Fibonacci sequence which is defined as follows:
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)

Implement a brute force recursive version and a memoized version. In the memoized version, you can store the elements of the sequence in any data structure with efficient access capability. Include a counter of the basic operations (recursive calls). How many recursive calls are made by the non-memoized brute force version for computing Fib(30)?

Compare the two versions (based on the resulting counts). Briefly explain why the brute force algorithm needs to use some form of dynamic programming?

B. Implement an inefficient, brute force, recursive algorithm to compute the Longest Common Subsequence of two strings (LCS). Copy this implementation and memoize it. Run both implementations on small input sequences and compare the running times (include a counter to count basic operations and print it after computing both versions of LCS).

Do the following on paper and hand in next week:
C. Based on the characterizing equations only, give a recursive algorithm for the 0-1 knapsack problem and memoize it so it is efficient. Compare your algorithm to the one given in the lecture notes in terms of time and space complexity.

D. Suppose we have a set of objects that have different sizes $s_1, s_2, \ldots, s_n$, and we have some positive upper limit L. Design an efficient algorithm to determine the subset of objects that produces the largest sum of sizes that is no greater than L. If you have time, implement your program and count the operations. Hint: dynamic programming similar to 0-1 knapsack problem.

C-5.9 How can we modify the dynamic programming algorithm from simply computing the best benefit value for the 0-1 knapsack problem to computing the assignment that gives this benefit?