Answersheet:
Algorithms

1. Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the problem can be divided into overlapping subproblems that can be solved independently. The solutions to these subproblems are stored to avoid redundant computations.

   Example: The Fibonacci sequence is a classic example where dynamic programming can be applied. The nth Fibonacci number can be defined as:
   $F(n) = F(n-1) + F(n-2)$ with base cases $F(0) = 0$ and $F(1) = 1$.

   Using dynamic programming, we can store the results of $F(n-1)$ and $F(n-2)$ to compute $F(n)$ efficiently.

2. A greedy algorithm makes a series of choices, each of which looks best at the moment, with the hope of finding a global optimum. It does not reconsider its choices, even if they lead to a suboptimal solution.

   Dynamic programming, on the other hand, solves problems by combining the solutions to subproblems. It is used when the problem can be broken down into overlapping subproblems, and it stores the results of these subproblems to avoid redundant computations.

   Example:

   - Greedy Algorithm: The coin change problem where the goal is to make change for a given amount using the fewest coins possible. A greedy algorithm might choose the largest denomination coin first.
   - Dynamic Programming: The same coin change problem can be solved using dynamic programming by considering all possible combinations of coins and storing the results of subproblems.

3. Memoization is a technique used in dynamic programming to store the results of expensive function calls and reuse them when the same inputs occur again. This helps in reducing the time complexity of algorithms by avoiding redundant computations.

   In the context of dynamic programming, memoization involves storing the results of subproblems in a data structure (like an array or a hash table) so that they can be quickly retrieved when needed.

   Example: In the Fibonacci sequence problem, instead of recalculating the Fibonacci number for each recursive call, we store the results of each Fibonacci number in an array and use these stored values to compute the next numbers.

4. The time complexity of the dynamic programming approach to solving the Fibonacci sequence is $O(n)$.

   This is because, in the dynamic programming approach, we compute each Fibonacci number once and store it in an array. Each computation takes constant time, and we perform a total of n computations, leading to a time complexity of $O(n)$.

   In contrast, the naive recursive approach has an exponential time complexity of $O(2^n)$ because it involves a lot of redundant calculations.

5. The longest common subsequence (LCS) problem can be solved using dynamic programming by breaking it down into smaller subproblems. The idea is to build a 2D table where the cell $dp[i][j]$ represents the length of the LCS of the first i characters of string X and the first j characters of string Y.

   The recurrence relation for filling the table is as follows:

   1. If the characters $X[i-1]$ and $Y[j-1]$ are the same, then $dp[i][j] = dp[i-1][j-1] + 1$.
   2. If the characters are different, then $dp[i][j] = extmax(dp[i-1][j], dp[i][j-1])$.

   The base cases are:

   - $dp[i][0] = 0$ for all i (an empty string has an LCS of length 0 with any string).
   - $dp[0][j] = 0$ for all j (an empty string has an LCS of length 0 with any string).

   By filling up the table using the above recurrence relations, we can find the length of the LCS of X and Y in $dp[m][n]$, where m and n are the lengths of X and Y, respectively.