|  |
| --- |
| //FIBONACCI  class Fibonacci {    // Function to calculate the nth Fibonacci number using  // recursion  static int nthFibonacci(int n){  // Base case: if n is 0 or 1, return n  if (n <= 1) {  return n;  }  // Recursive case: sum of the two preceding  // Fibonacci numbers  return nthFibonacci(n - 1) + nthFibonacci(n - 2);  }  public static void main(String[] args){  int n = 5;  int result = nthFibonacci(n);  System.out.println(result);  }  } |
| // MEMOIZATION  import java.util.Arrays;  class Memoization {  // Function to calculate the nth Fibonacci number using memoization  static int nthFibonacciUtil(int n, int[] memo) {    // Base case: if n is 0 or 1, return n  if (n <= 1) {  return n;  }  // Check if the result is already in the memo table  if (memo[n] != -1) {  return memo[n];  }  // Recursive case: calculate Fibonacci number  // and store it in memo  memo[n] = nthFibonacciUtil(n - 1, memo)  + nthFibonacciUtil(n - 2, memo);  return memo[n];  }  // Wrapper function that handles both initialization  // and Fibonacci calculation  static int nthFibonacci(int n) {  // Create a memoization table and initialize with -1  int[] memo = new int[n + 1];  Arrays.fill(memo, -1);    // Call the utility function  return nthFibonacciUtil(n, memo);  }  public static void main(String[] args) {  int n = 5;  int result = nthFibonacci(n);  System.out.println(result);  }  } |
| // TABULATION  class Tabulation {    // Function to calculate the nth Fibonacci number using iteration  static int nthFibonacci(int n) {  // Handle the edge cases  if (n <= 1) return n;    // Create an array to store Fibonacci numbers  int[] dp = new int[n + 1];  // Initialize the first two Fibonacci numbers  dp[0] = 0;  dp[1] = 1;  // Fill the array iteratively  for (int i = 2; i <= n; ++i) {  dp[i] = dp[i - 1] + dp[i - 2];  }  // Return the nth Fibonacci number  return dp[n];  }  public static void main(String[] args) {  int n = 5;  int result = nthFibonacci(n);  System.out.println(result);  }  } |
| **//LCSMemoization:**  public class LCSMemoization {  public static int lcs(String s1, String s2, int m, int n, int[][] memo) {  if (m == 0 || n == 0) return 0;  if (memo[m][n] != -1) return memo[m][n];  if (s1.charAt(m - 1) == s2.charAt(n - 1)) {  memo[m][n] = 1 + lcs(s1, s2, m - 1, n - 1, memo);  } else {  memo[m][n] = Math.max(  lcs(s1, s2, m - 1, n, memo),  lcs(s1, s2, m, n - 1, memo)  );  }  return memo[m][n];  }  public static void main(String[] args) {  String s1 = "abcde";  String s2 = "ace";  int m = s1.length(), n = s2.length();  int[][] memo = new int[m + 1][n + 1];  for (int[] row : memo) java.util.Arrays.fill(row, -1);  System.out.println("LCS length: " + lcs(s1, s2, m, n, memo)); // Output: 3  }  } |
| **//LCSTabulation**:  public class LCSTabulation {  public static int longestCommonSubsequence(String s1, String s2) {  int m = s1.length(), n = s2.length();  int[][] dp = new int[m + 1][n + 1];  for (int i = 1; i <= m; i++) {  for (int j = 1; j <= n; j++) {  if (s1.charAt(i - 1) == s2.charAt(j - 1))  dp[i][j] = 1 + dp[i - 1][j - 1];  else  dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);  }  }  return dp[m][n];  }  public static void main(String[] args) {  String s1 = "abcde";  String s2 = "ace";  System.out.println("LCS length: " + longestCommonSubsequence(s1, s2)); // Output: 3  }  } |
| **// KnapsackMemoization**:  public class KnapsackMemoization {  public static int knapsack(int[] weights, int[] values, int capacity, int n, Integer[][] memo) {  if (n == 0 || capacity == 0) return 0;  if (memo[n][capacity] != null) return memo[n][capacity];  if (weights[n - 1] <= capacity) {  int include = values[n - 1] + knapsack(weights, values, capacity - weights[n - 1], n - 1, memo);  int exclude = knapsack(weights, values, capacity, n - 1, memo);  memo[n][capacity] = Math.max(include, exclude);  } else {  memo[n][capacity] = knapsack(weights, values, capacity, n - 1, memo);  }  return memo[n][capacity];  }  public static void main(String[] args) {  int[] weights = {2, 3, 4, 5};  int[] values = {3, 4, 5, 6};  int capacity = 5;  int n = weights.length;  Integer[][] memo = new Integer[n + 1][capacity + 1];  System.out.println("Max value: " + knapsack(weights, values, capacity, n, memo)); // Output: 7  }  } |
| **//KnapsackTabulation**:  public class KnapsackTabulation {  public static int knapsack(int[] weights, int[] values, int capacity) {  int n = weights.length;  int[][] dp = new int[n + 1][capacity + 1];  for (int i = 1; i <= n; i++) {  for (int w = 1; w <= capacity; w++) {  if (weights[i - 1] <= w) {  dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);  } else {  dp[i][w] = dp[i - 1][w];  }  }  }  return dp[n][capacity];  }  public static void main(String[] args) {  int[] weights = {2, 3, 4, 5};  int[] values = {3, 4, 5, 6};  int capacity = 5;  System.out.println("Max value: " + knapsack(weights, values, capacity)); // Output: 7  }  } |
| // |
|  |