Dynamic programming

70. Climbing Stairs

1. You are climbing a staircase. It takes n steps to reach the top.
2. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution :

```
class Solution {
   public int climbStairs(int n) {
        int[] res = new int[n+1];
        res[0]=0;
        res[1] = 1;
        res[2] = 2;
        for (int i = 3; i< n+1;i++) {
            res[i] = res[i-1] + res[i-2];
        }
        return res[n];
   }
}
```

72. Edit Distance

Given two strings word1 and word2, return *the minimum number of operations required to convert word1 to word2*.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

```
public class Solution {
   public int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();

        // Initialize a 2D DP array with dimensions (m+1) x (n+1)
```

```
        int[][] dp = new int[m + 1][n + 1];

        // Base case: when one of the words is empty, the number of operations
needed
        // is equal to the length of the other word
        for (int i = 0; i <= m; i++) {
            dp[i][0] = i;
        }
        for (int j = 0; j <= n; j++) {
            dp[0][j] = j;
        }

        // Fill in the DP table
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // If the characters at the current positions are equal, no
operation needed
                if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    // Otherwise, choose the minimum of the three possible
operations:
                    // 1. Insertion (dp[i][j - 1] + 1)
                    // 2. Deletion (dp[i - 1][j] + 1)
                    // 3. Replacement (dp[i - 1][j - 1] + 1)
                    dp[i][j] = Math.min(Math.min(dp[i][j - 1], dp[i - 1][j]),
dp[i - 1][j - 1]) + 1;
                }
            }
        }

        // The minimum number of operations needed to convert word1 to word2 is
stored in dp[m][n]
        return dp[m][n];
    }
}
```

188. Best Time to Buy and Sell Stock IV

You are given an integer array prices where prices[i] is the price of a given stock on the i^{th} day, and an integer k.

Find the maximum profit you can achieve. You may complete at most k transactions: i.e. you may buy at most k times and sell at most k times.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again)

Example 1:

Input: k = 2, prices = [2,4,1]
Output: 2
Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.

```
public class Solution {
   public int maxProfit(int k, int[] prices) {
        int n = prices.length;
        if (n <= 1 || k == 0) {
            return 0;
        }

        if (k >= n / 2) {
            // If k is large enough, it's equivalent to the problem of infinite
transactions
            int maxProfit = 0;
            for (int i = 1; i < n; i++) {
                if (prices[i] > prices[i - 1]) {
                    maxProfit += prices[i] - prices[i - 1];
                }
            }
            return maxProfit;
        }

        // Create a 2D array to store the maximum profit for each transaction
and day
        int[][] dp = new int[k + 1][n];
```

```
        // Calculate the maximum profit using dynamic programming
        for (int i = 1; i <= k; i++) {
            int maxDiff = -prices[0];
            for (int j = 1; j < n; j++) {
                dp[i][j] = Math.max(dp[i][j - 1], prices[j] + maxDiff);
                maxDiff = Math.max(maxDiff, dp[i - 1][j] - prices[j]);
            }
        }

        return dp[k][n - 1];
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int k = 2;
        int[] prices = {2, 4, 1};
        System.out.println(solution.maxProfit(k, prices)); // Output: 2
    }
}
```

322. Coin Change

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

```
public int coinChange(int[] coins, int amount) {// Create an array to store the fewest number of coins needed to make up each amount

        int[] dp = new int[amount + 1];

        Arrays.fill(dp, amount + 1);

        dp[0] = 0; // Base case: 0 coins needed to make up 0 amount
```

```
        // Iterate through each amount from 1 to the target amount
        for (int i = 1; i <= amount; i++) {
            // Try each coin denomination
            for (int coin : coins) {
                if (coin <= i)
// If using this coin results in a smaller number of coins needed, update dp[i]
                    dp[i] = Math.min(dp[i], dp[i - coin] + 1);
                }
            }
        }
        // If dp[amount] is still amount + 1, it means the amount cannot be made up by any combination of coins
        return dp[amount] > amount ? -1 : dp[amount];
    }
```

300. Longest Increasing Subsequence

Solved

Medium

TopicsCompanies

Given an integer array nums, return *the length of the longest **strictly increasing***

subsequence

Example 1:

Input: nums = [10,9,2,5,3,7,101,18]

Output: 4

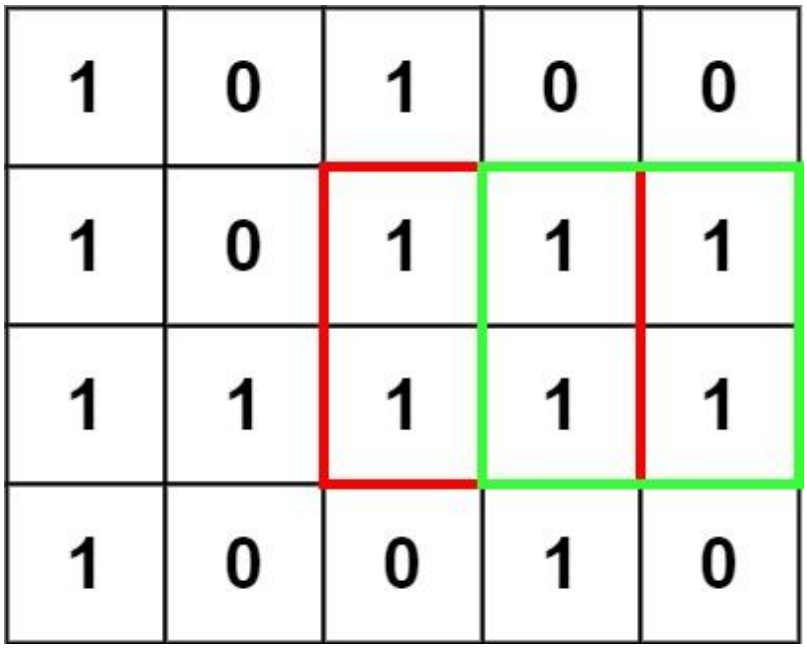
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
```

```
        }

        int[] dp = new int[nums.length];
        Arrays.fill(dp, 1);
        int maxLength = 1;

        for (int i = 1; i < nums.length; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                    maxLength = Math.max(maxLength, dp[i]);
                }
            }
        }

        return maxLength;
    }
}
```

Given an m x n binary matrix filled with 0's and 1's, *find the largest square containing only* 1's *and return its area*.

Example 1:



Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 4

```
public class Solution {
   public int maximalSquare(char[][] matrix) {
       if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
           return 0;
       }

       int m = matrix.length;
       int n = matrix[0].length;

       // Create a 2D array to store the maximum size of square ending at each
position
       int[][] dp = new int[m][n];

       int maxSquareSize = 0;

       // Fill the first row and column of dp array with the values from matrix
```

```
        for (int i = 0; i < m; i++) {
            dp[i][0] = matrix[i][0] - '0';
            maxSquareSize = Math.max(maxSquareSize, dp[i][0]);
        }
        for (int j = 0; j < n; j++) {
            dp[0][j] = matrix[0][j] - '0';
            maxSquareSize = Math.max(maxSquareSize, dp[0][j]);
        }

        // Iterate through the matrix and fill the dp array
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][j] == '1') {
                    // Calculate the size of square ending at position (i, j)
                    dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j],
dp[i][j - 1])) + 1;
                    maxSquareSize = Math.max(maxSquareSize, dp[i][j]);
                }
            }
        }

        return maxSquareSize * maxSquareSize;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        char[][] matrix = {
            {'1', '0', '1', '0', '0'},
            {'1', '0', '1', '1', '1'},
            {'1', '1', '1', '1', '1'},
            {'1', '0', '0', '1', '0'}
        };
        System.out.println(solution.maximalSquare(matrix)); // Output: 4
    }
}
```

139. Word Break

Solved

Medium

TopicsCompanies

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

```
public class Solution {
        public boolean wordBreak(String s, List<String> wordDict) {
            // Convert wordDict to a set for faster lookup
            Set<String> wordSet = new HashSet<>(wordDict);

            // Create a boolean array to track if a substring can be segmented
            boolean[] dp = new boolean[s.length() + 1];
            dp[0] = true;

            // Iterate through the string
            for (int i = 1; i <= s.length(); i++) {
                // Check if the substring from index 0 to i can be segmented
                for (int j = 0; j < i; j++) {
                    if (dp[j] && wordSet.contains(s.substring(j, i))) {
                        dp[i] = true;
                        break;
                    }
                }
            }
            return dp[s.length()];
    }
}
```

2-D DP

64. Minimum Path Sum

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

```
public class Solution {
    public int minPathSum(int[][] grid) {
```

```
        int m = grid.length; // Number of rows
        int n = grid[0].length; // Number of columns

        // Initialize a 2D array to store the minimum sum to reach each cell
        int[][] dp = new int[m][n];

        // Base case: Initialize the first cell with the value from the grid
        dp[0][0] = grid[0][0];

        // Initialize the first row: sum of values to reach each cell
        for (int j = 1; j < n; j++) {
            dp[0][j] = dp[0][j - 1] + grid[0][j];
        }

        // Initialize the first column: sum of values to reach each cell
        for (int i = 1; i < m; i++) {
            dp[i][0] = dp[i - 1][0] + grid[i][0];
        }

        // Compute the minimum sum to reach each cell
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                // Minimum sum to reach the current cell is the sum of the value in the current cell
                // and the minimum of the values to reach the cell above and the cell to the left
                dp[i][j] = grid[i][j] + Math.min(dp[i - 1][j], dp[i][j - 1]);
            }
        }

        // Return the minimum sum to reach the bottom-right cell
        return dp[m - 1][n - 1];
    }
}
```

63. Unique Paths II

Solved

Medium

TopicsCompanies

Hint

You are given an m x n integer array grid. There is a robot initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

An obstacle and space are marked as 1 or 0 respectively in grid. A path that the robot takes cannot include **any** square that is an obstacle.

Return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

```
public class Solution {
   public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length; // Number of rows
        int n = obstacleGrid[0].length; // Number of columns

        // Initialize a 2D array to store the number of unique paths to reach
each cell
        int[][] dp = new int[m][n];

        // Base case: Initialize the first cell with 1 if it's not an obstacle,
0 otherwise
        dp[0][0] = (obstacleGrid[0][0] == 0) ? 1 : 0;

        // Initialize the first row: number of unique paths to reach each cell
        for (int j = 1; j < n; j++) {
            if (obstacleGrid[0][j] == 0) {
                dp[0][j] = dp[0][j - 1];
            }
        }

        // Initialize the first column: number of unique paths to reach each
cell
        for (int i = 1; i < m; i++) {
            if (obstacleGrid[i][0] == 0) {
                dp[i][0] = dp[i - 1][0];
            }
        }

        // Compute the number of unique paths to reach each cell
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                // If the current cell is not an obstacle, compute the number of
unique paths
                if (obstacleGrid[i][j] == 0) {
                    dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
                }
```

```
            }
        }

        // Return the number of unique paths to reach the bottom-right corner
        return dp[m - 1][n - 1];
    }
}
```

5. Longest Palindromic Substring

Given a string s, return *the longest Palindromic substring* in s

Example 1:

Input: s = "babad"

Output: "bab"

Explanation: "aba" is also a valid answer

Example 2:

Input: s = "cbbd"

Output: "bb"

```
public class Solution {
    public String longestPalindrome(String s) {
        int n = s.length();
        boolean[][] dp = new boolean[n][n];
        String longestPalindrome = "";

        // Base case: single characters are palindrome
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
            longestPalindrome = s.substring(i, i + 1);
        }

        // Check for palindromic substrings of length 2
        for (int i = 0; i < n - 1; i++) {
```

```
            if (s.charAt(i) == s.charAt(i + 1)) {
                dp[i][i + 1] = true;
                longestPalindrome = s.substring(i, i + 2);
            }
        }

        // Check for palindromic substrings of length greater than 2
        //here dp is based on length of the string considered for palindrome and
is made of 2 d array of that string length.
        for (int length = 3; length <= n; length++) {
            for (int i = 0; i <= n - length; i++) {
                int j = i + length - 1;
                // because i and j chars are length apart and are checked for
equality to check for palindrome
                //checking if i and j chars are same and dp in between is sorted
already.
                if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
                    dp[i][j] = true;
                    longestPalindrome = s.substring(i, j + 1);
                }
            }
        }

        return longestPalindrome;
    }
}
```

97. Interleaving String

Given strings s1, s2, and s3, find whether s3 is formed by an **interleaving** of s1 and s2.

An **interleaving** of two strings s and t is a configuration where s and t are divided into n and m

substrings

respectively, such that:

- $s = s_1 + s_2 + ... + s_n$
- $t = t_1 + t_2 + ... + t_m$
- $|n - m| <= 1$
- The **interleaving** is $s_1 + t_1 + s_2 + t_2 + s_3 + t_3 + ...$ or $t_1 + s_1 + t_2 + s_2 + t_3 + s_3 + ...$

Note: a + b is the concatenation of strings a and b.

Note : here it is such that S3 len = s1 + s2 length.

```
public class Solution {
   public boolean isInterleave(String s1, String s2, String s3) {
        int m = s1.length(), n = s2.length();
        if (m + n != s3.length()) return false;

        boolean[][] dp = new boolean[m + 1][n + 1];

        // Initialize dp[0][0] as true
        dp[0][0] = true;

        // Fill the first row
        for (int j = 1; j <= n; j++) {
            dp[0][j] = dp[0][j - 1] && s2.charAt(j - 1) == s3.charAt(j - 1);
        }

        // Fill the first column
        for (int i = 1; i <= m; i++) {
            dp[i][0] = dp[i - 1][0] && s1.charAt(i - 1) == s3.charAt(i - 1);
        }

        // Fill the remaining cells
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                dp[i][j] = (dp[i - 1][j] && s1.charAt(i - 1) == s3.charAt(i + j
- 1))
                        || (dp[i][j - 1] && s2.charAt(j - 1) == s3.charAt(i + j
- 1));
            }
        }

        return dp[m][n];
    }
}
```