```
package Love_Babbar;
public class Revision{
	//---Coin Change Problem
	//---Knapsack Problem
	//---Binomial Coefficient Problem
	//Permutation Coefficient Problem
	//Program for nth Catalan Number
	//Matrix Chain Multiplication
	//Edit Distance
	//Subset Sum Problem
	//Friends Pairing Problem
	//Gold Mine Problem
	//Assembly Line Scheduling Problem
	//Painting the Fence problem
	//Maximize The Cut Segments
	//Longest Common Subsequence
	//Longest Repeated Subsequence
	//Longest Increasing Subsequence
	//Space Optimized Solution of LCS
	//LCS (Longest Common Subsequence) of three strings
	//Maximum Sum Increasing Subsequence
	//Count all subsequences having product less than K
	//Longest subsequence such that difference between adjacent is one
	//Maximum subsequence sum such that no three are consecutive
	//Egg Dropping Problem
	//Maximum Length Chain of Pairs
	//Maximum size square sub-matrix with all 1s
	//Maximum sum of pairs with specific difference
	//Min Cost Path Problem
	//Maximum difference of zeros and ones in binary string
	//Minimum number of jumps to reach end
	//Minimum cost to fill given weight in a bag
	//Minimum removals from array to make max –min <= K
	//Longest Common Substring
	//Count number of ways to reach a given score in a game
	//Count Balanced Binary Trees of Height h
	//LargestSum Contiguous Subarray [V>V>V>V IMP ]
	//Smallest sum contiguous subarray
	//Unbounded Knapsack (Repetition of items allowed)
	//Word Break Problem
	//Largest Independent Set Problem
	//Partition problem
	//Longest Palindromic Subsequence
	//Count All Palindromic Subsequence in a given String
	//Longest Palindromic Substring
	//Longest alternating subsequence
	//Weighted Job Scheduling
	//Coin game winner where every player has three choices
	//Count Derangements (Permutation such that no element appears in its original
```

```
position) [ IMPORTANT ]
        //Maximum profit by buying and selling a share at most twice [ IMP ]
        //Optimal Strategy for a Game
        //Optimal Binary Search Tree
        //Palindrome Partitioning Problem
        //Word Wrap Problem
        //Mobile Numeric Keypad Problem [ IMP ]
        //Boolean Parenthesization Problem
        //Largest rectangular sub-matrix whose sum is 0
        //Largest area rectangular sub-matrix with equal number of 1's and 0's [ IMP ]
        //Maximum sum rectangle in a 2D matrix
        //Maximum profit by buying and selling a share at most k times
        //Find if a string is interleaved of two other strings
        //Maximum Length of Pair Chain
}
public class DP {
    class DP_LB {
        // Coin Change Problem
        class Solution {
            public long count(int coins[], int N, int sum) {
                // code here.
                long dp[][] = new long[N + 1][sum + 1];
                for (long row[] : dp) {
                    Arrays.fill(row, 0);
                }
                for (int i = 0; i < N + 1; i++) {
                    dp[i][0] = 1;
                }
                for (int ind = N - 1; ind >= 0; ind--) {
                    for (int target = 0; target <= sum; target++) {
                        long notTake = dp[ind + 1][target];
                        long take = 0;
                        if (coins[ind] <= target) {
                            take = dp[ind][target - coins[ind]];
                        }
                        dp[ind][target] = take + notTake;
                    }
                }
                return dp[0][sum];
                // return f(0,sum,coins,dp,N);
            }

            public int f(int ind, int target, int coins[], int dp[][], int N) {
                // Base Case
                if (target == 0) {
                    return 1;
                }
                if (ind == N) {
                    return 0;
```

```
        }
        if (dp[ind][target] != -1) {
            return dp[ind][target];
        }
        // Recursion
        int notTake = f(ind + 1, target, coins, dp, N);
        int take = 0;
        if (coins[ind] <= target) {
            take = f(ind, target - coins[ind], coins, dp, N);
        }
        return dp[ind][target] = take + notTake;
    }
}

// Knapsack Problem
class Solution {
    // Function to return max value that can be put in knapsack of capacity W.
    static int knapSack(int WT, int wt[], int val[], int n) {
        // your code here
        int dp[][] = new int[n + 1][WT + 1];
        for (int row[] : dp) {
            Arrays.fill(row, 0);
        }
        for (int ind = n - 1; ind >= 0; ind--) {
            for (int target = 0; target <= WT; target++) {
                int notTake = dp[ind + 1][target];
                int take = 0;
                if (wt[ind] <= target) {
                    take = val[ind] + dp[ind + 1][target - wt[ind]];
                }
                dp[ind][target] = Math.max(take, notTake);
            }
        }
        return dp[0][WT];
        // return f(0,WT,wt,val,n,dp);
    }

    static int f(int ind, int W, int wt[], int val[], int n, int dp[][]) {
        if (W == 0) {
            return 0;
        }
        if (ind == n) {
            return 0;
        }
        if (dp[ind][W] != -1) {
            return dp[ind][W];
        }

        int notTake = f(ind + 1, W, wt, val, n, dp);
```

```
        int take = 0;
        if (wt[ind] <= W) {
            take = val[ind] + f(ind + 1, W - wt[ind], wt, val, n, dp);
        }
        return dp[ind][W] = Math.max(take, notTake);
    }
}

// Binomial Coefficient Problem
class Solution {
    static int nCr(int n, int r) {
        // code here
        int dp[][] = new int[n + 1][r + 1];
        for (int row[] : dp) {
            Arrays.fill(row, 0);
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < r; j++) {
                if (j == 0) {
                    dp[i][j] = 1;
                    continue;
                }
                if (j == i) {
                    dp[i][j] = 1;
                    continue;
                }
                if (j > i) {
                    dp[i][j] = 0;
                }
            }
        }

        for (int N = 1; N <= n; N++) {
            for (int K = 1; K <= r; K++) {
                int one = dp[N - 1][K - 1];
                int two = dp[N - 1][K];
                dp[N][K] = (one + two) % ((int) (1e9) + 7);
            }
        }
        return dp[n][r];
        // return f(r,n,dp);
    }

    static int f(int k, int n, int dp[][]) {
        if (k > n) {
            return 0;
        }
        if (k == 0 || k == n) {
            return 1;
```

```
            }
            if (dp[n][k] != -1) {
                return dp[n][k];
            }
            int one = f(k - 1, n - 1, dp);
            int two = f(k, n - 1, dp);
            return dp[n][k] = (one + two) % ((int) (1e9) + 7);
        }
    }

    // Permutation Coefficient Problem
    // Program for nth Catalan Number
    // Matrix Chain Multiplication
    class Solution {
        static int f(int arr[], int i, int j, int[][] dp) {
            if (i == j)
                return 0;
            if (dp[i][j] != -1)
                return dp[i][j];
            int mini = Integer.MAX_VALUE;
            for (int k = i; k <= j - 1; k++) {
                int ans = f(arr, i, k, dp) + f(arr, k + 1, j, dp) + arr[i - 1] * arr[k] * arr[j];
                mini = Math.min(mini, ans);
            }
            return mini;
        }
    }

    // Edit Distance
    class Solution {
        static int editDistanceUtil(String S1, String S2, int i, int j, int[][] dp) {
            if (i < 0)
                return j + 1;
            if (j < 0)
                return i + 1;
            if (dp[i][j] != -1)
                return dp[i][j];
            if (S1.charAt(i) == S2.charAt(j))
                return dp[i][j] = 0 + editDistanceUtil(S1, S2, i - 1, j - 1, dp);
            // Minimum of three choices
            else
                return dp[i][j] = 1 + Math.min(editDistanceUtil(S1, S2, i - 1, j - 1, dp),
                        Math.min(editDistanceUtil(S1, S2, i - 1, j, dp), editDistanceUtil(S1, S2,
i, j - 1, dp)));
            }

        static int editDistance(String S1, String S2) {
            int n = S1.length();
            int m = S2.length();
```

```
        int[][] dp = new int[n + 1][m + 1];
        for (int i = 0; i <= n; i++) {
            dp[i][0] = i;
        }
        for (int j = 0; j <= m; j++) {
            dp[0][j] = j;
        }
        for (int i = 1; i < n + 1; i++) {
            for (int j = 1; j < m + 1; j++) {
                if (S1.charAt(i - 1) == S2.charAt(j - 1))
                    dp[i][j] = 0 + dp[i - 1][j - 1];
                else
                    dp[i][j] = 1 + Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1]));
            }
        }
        return dp[n][m];
    }
}

// Subset Sum Problem
class Solution {
    class TUF {
        static boolean subsetSumUtil(int ind, int target, int[] arr, int[][] dp) {
            if (target == 0)
                return true;
            if (ind == 0)
                return arr[0] == target;
            if (dp[ind][target] != -1)
                return dp[ind][target] == 0 ? false : true;
            boolean notTaken = subsetSumUtil(ind - 1, target, arr, dp);
            boolean taken = false;
            if (arr[ind] <= target)
                taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);
            dp[ind][target] = notTaken || taken ? 1 : 0;
            return notTaken || taken;
        }
    }

    class TUF {
        static boolean subsetSumToK(int n, int k, int[] arr) {
            boolean dp[][] = new boolean[n][k + 1];
            for (int i = 0; i < n; i++) {
                dp[i][0] = true;
            }
            if (arr[0] <= k)
                dp[0][arr[0]] = true;
            for (int ind = 1; ind < n; ind++) {
                for (int target = 1; target <= k; target++) {
                    boolean notTaken = dp[ind - 1][target];
```

```
                boolean taken = false;
                if (arr[ind] <= target)
                    taken = dp[ind - 1][target - arr[ind]];
                dp[ind][target] = notTaken || taken;
            }
        }
        return dp[n - 1][k];
    }
  }
}

// Friends Pairing Problem
class Solution {
    public long countFriendsPairings(int n) {
        // code here
        long dp[] = new long[n + 1];
        Arrays.fill(dp, 0);
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            long single = dp[i - 1];
            long doubl = dp[i - 2] * (i - 1);
            dp[i] = (single + doubl) % (int) (1e9 + 7);
        }
        return dp[n];
        // return f(n,dp);

    }

    public long f(int n, long dp[]) {
        if (n == 0 || n == 1) {
            return 1;
        }
        if (dp[n] != -1) {
            return dp[n];
        }
        long single = f(n - 1, dp);
        long doubl = f(n - 2, dp) * (n - 1);
        return dp[n] = (single + doubl) % (int) (1e9 + 7);
    }
}

// Gold Mine Problem
class Solution {
    static int maxGold(int n, int m, int M[][]) {
        // code here
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int dp[][] = new int[n + 1][m + 1];
```

```
            for (int row[] : dp) {
                Arrays.fill(row, -1);
            }
            ans = Math.max(ans, f(i, 0, M, n, m, dp));
        }
        return ans;
    }

    public static int f(int row, int col, int mat[][], int n, int m, int dp[][]) {

        if (row < 0 || row > n - 1) {
            return -(int) (1e9);
        }
        if (col == m - 1 && row >= 0 && row < n) {
            return mat[row][col];
        }
        if (dp[row][col] != -1) {
            return dp[row][col];
        }
        int rightUp = mat[row][col] + f(row - 1, col + 1, mat, n, m, dp);
        int right = mat[row][col] + f(row, col + 1, mat, n, m, dp);
        int rightDown = mat[row][col] + f(row + 1, col + 1, mat, n, m, dp);
        return dp[row][col] = Math.max(rightUp, Math.max(right, rightDown));
    }
}

// Assembly Line Scheduling Problem
// Painting the Fence problem
// Maximize The Cut Segments
class Solution {
    // Function to find the maximum number of cuts.
    public int maximizeCuts(int n, int x, int y, int z) {
        // Your code here
        int dp[][] = new int[3][n + 1];
        for (int row[] : dp) {
            Arrays.fill(row, 0);
        }
        int arr[] = new int[3];
        arr[0] = x;
        arr[1] = y;
        arr[2] = z;
        return f(0, n, dp, 3, arr);
    }

    public int f(int ind, int len, int dp[][], int N, int arr[]) {
        // Base Case
        if (len == 0) {
            return 0;
        }
```

```
        if (len < 0) {
            return -(int) (1e9);
        }
        if (ind == N) {
            return -(int) (1e9);
        }
        // Recursive Case
        int notTake = f(ind + 1, len, dp, N, arr);
        int take = 0;
        if (arr[ind] <= len) {
            take = 1 + f(ind, len - arr[ind], dp, N, arr);
        }
        return dp[ind][len] = Math.max(take, notTake);
    }
}

// Longest Common Subsequence
class Solution {
    // Function to find the length of longest common subsequence in two strings.
    static int lcs(int x, int y, String s1, String s2) {
        // your code here
        int dp[][] = new int[x + 1][y + 1];
        for (int rows[] : dp) {
            Arrays.fill(rows, -1);
        }
        for (int i = 0; i <= x; i++) {
            dp[i][0] = 0;
        }
        for (int i = 0; i <= y; i++) {
            dp[0][i] = 0;
        }

        for (int ind1 = 1; ind1 <= x; ind1++) {
            for (int ind2 = 1; ind2 <= y; ind2++) {
                if (s1.charAt(ind1 - 1) == s2.charAt(ind2 - 1)) {
                    dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
                } else {
                    dp[ind1][ind2] = 0 + Math.max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
                }
            }
        }
        int i = x;
        int j = y;
        String ans = "";
        while (i > 0 && j > 0) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                ans += s1.charAt(i - 1) + "";
                i--;
                j--;
```

```
                } else if (dp[i - 1][j] > dp[i][j - 1]) {
                    i--;
                } else {
                    j--;
                }
            }
            System.out.println(ans);
            return dp[x][y];

        }

        static int f(String s1, String s2, int ind1, int ind2, int dp[][]) {
            if (ind1 == 0 || ind2 == 0)
                return 0;
            if (dp[ind1][ind2] != -1)
                return dp[ind1][ind2];
            // Chararcter Match
            if (s1.charAt(ind1 - 1) == s2.charAt(ind2 - 1)) {
                return dp[ind1][ind2] = 1 + f(s1, s2, ind1 - 1, ind2 - 1, dp);
            } else {
                return dp[ind1][ind2] = Math.max(f(s1, s2, ind1 - 1, ind2, dp), f(s1, s2, ind1,
ind2 - 1, dp));
            }
        }

    }

    // Longest Repeated Subsequence
    class Solution {
        public int LongestRepeatingSubsequence(String str) {
            // code here
            int n = str.length();
            int dp[][] = new int[n + 1][n + 1];
            for (int rows[] : dp) {
                Arrays.fill(rows, -1);
            }
            return f(str, str, n - 1, n - 1, dp);
        }

        public int f(String s1, String s2, int ind1, int ind2, int dp[][]) {

            if (ind1 < 0 || ind2 < 0) {
                return 0;
            }
            if (dp[ind1][ind2] != -1) {
                return dp[ind1][ind2];
            }
            if (s1.charAt(ind1) == s2.charAt(ind2) && ind1 != ind2) {
                return dp[ind1][ind2] = 1 + f(s1, s2, ind1 - 1, ind2 - 1, dp);
```

```
            }
            return dp[ind1][ind2] = Math.max(f(s1, s2, ind1 - 1, ind2, dp), f(s1, s2, ind1,
ind2 - 1, dp));
        }
    }

    // Longest Increasing Subsequence
    class Solution {
        class TUF {
            static int longestIncreasingSubsequence(int arr[], int n) {
                int dp[][] = new int[n + 1][n + 1];
                for (int ind = n - 1; ind >= 0; ind--) {
                    for (int prev_index = ind - 1; prev_index >= -1; prev_index--) {
                        int notTake = 0 + dp[ind + 1][prev_index + 1];
                        int take = 0;
                        if (prev_index == -1 || arr[ind] > arr[prev_index]) {
                            take = 1 + dp[ind + 1][ind + 1];
                        }
                        dp[ind][prev_index + 1] = Math.max(notTake, take);
                    }
                }
                return dp[0][0];
            }

            static int longestIncreasingSubsequence(int arr[], int n) {
                int dp[] = new int[n];
                Arrays.fill(dp, 1);
                for (int i = 0; i <= n - 1; i++) {
                    for (int prev_index = 0; prev_index <= i - 1; prev_index++) {
                        if (arr[prev_index] < arr[i]) {
                            dp[i] = Math.max(dp[i], 1 + dp[prev_index]);
                        }
                    }
                }
                int ans = -1;
                for (int i = 0; i <= n - 1; i++) {
                    ans = Math.max(ans, dp[i]);
                }
                return ans;
            }

            static int longestIncreasingSubsequence(int arr[], int n) {
                int[] dp = new int[n];
                Arrays.fill(dp, 1);
                int[] hash = new int[n];
                Arrays.fill(hash, 1);

                for (int i = 0; i <= n - 1; i++) {
                    hash[i] = i; // initializing with current index
```

```
                for (int prev_index = 0; prev_index <= i - 1; prev_index++) {
                    if (arr[prev_index] < arr[i] && 1 + dp[prev_index] > dp[i]) {
                        dp[i] = 1 + dp[prev_index];
                        hash[i] = prev_index;
                    }
                }
            }
            int ans = -1;
            int lastIndex = -1;
            for (int i = 0; i <= n - 1; i++) {
                if (dp[i] > ans) {
                    ans = dp[i];
                    lastIndex = i;
                }
            }
            ArrayList<Integer> temp = new ArrayList<>();
            temp.add(arr[lastIndex]);
            while (hash[lastIndex] != lastIndex) { // till not reach the initialization value
                lastIndex = hash[lastIndex];
                temp.add(arr[lastIndex]);
            }
            for (int i = temp.size() - 1; i >= 0; i--) {
                System.out.print(temp.get(i) + " ");
            }
            return ans;
        }
    }
}

// Space Optimized Solution of LCS
// LCS (Longest Common Subsequence) of three strings
// Maximum Sum Increasing Subsequence
class Solution {
    public int maxSumIS(int arr[], int n) {
        // code here.
        int i, j, max = 0;
        int dp[] = new int[n];
        for (i = 0; i < n; i++) {
            dp[i] = arr[i];
        }
        for (i = 1; i < n; i++) {
            for (j = 0; j < i; j++) {
                if (arr[i] > arr[j] && dp[i] < dp[j] + arr[i]) {
                    dp[i] = dp[j] + arr[i];
                }
            }
        }
        for (i = 0; i < n; i++) {
            if (max < dp[i]) {
```

```
                max = dp[i];
            }
        }
        return max;
    }
}

// Count all subsequences having product less than K
public class Solution {
    public static int countSubsequences(int[] a, int n, int p) {
        // Write your code here.
        int dp[][] = new int[p + 1][n];
        for (int rows[] : dp) {
            Arrays.fill(rows, -1);
        }
        return f(p, n - 1, a, dp) - 1;
    }

    public static int f(int product, int ind, int arr[], int dp[][]) {
        if (ind < 0) {
            return 1;
        }
        if (dp[product][ind] != -1) {
            return dp[product][ind];
        }
        int notTake = f(product, ind - 1, arr, dp);
        int take = 0;
        if (arr[ind] <= product) {
            take = f(product / arr[ind], ind - 1, arr, dp);
        }
        return dp[product][ind] = take + notTake;
    }
}

// Longest subsequence such that difference between adjacent is one
class Solution {
    static int longestSubsequence(int n, int arr[]) {
        // code here
        int dp[] = new int[n];
        Arrays.fill(dp, 1);
        int max = 1;
        for (int i = 0; i <= n - 1; i++) {
            for (int prev_index = 0; prev_index <= i - 1; prev_index++) {
                if (Math.abs(arr[i] - arr[prev_index]) == 1) {
                    dp[i] = Math.max(dp[i], 1 + dp[prev_index]);
                    max = Math.max(max, dp[i]);
                }
            }
        }
```

```
        return max;
    }
}

// Maximum subsequence sum such that no three are consecutive
class Solution {
    int findMaxSum(int arr[], int n) {
        // code here
        int sum[] = new int[n];
        Arrays.fill(sum, -1);
        return f(n - 1, sum, arr);
    }

    int f(int ind, int sum[], int arr[]) {
        if (sum[ind] != -1) {
            return sum[ind];
        }
        if (ind == 0) {
            return arr[0];
        }
        if (ind == 1) {
            return arr[0] + arr[1];
        }
        if (ind == 2) {
            return Math.max(arr[0] + arr[1], Math.max(arr[1] + arr[2], arr[0] + arr[2]));
        }
        ;
        int one = arr[ind] + f(ind - 2, sum, arr);
        int two = arr[ind] + arr[ind - 1] + f(ind - 3, sum, arr);
        int three = f(ind - 1, sum, arr);
        return sum[ind] = Math.max(one, Math.max(two, three));

    }
}

// Egg Dropping Problem
class Solution {
    // Function to find minimum number of attempts needed in
    // order to find the critical floor.
    static int eggDrop(int n, int k) {
        int dp[][] = new int[n + 1][k + 1];
        for (int rows[] : dp) {
            Arrays.fill(rows, 0);
        }
        for (int i = 0; i <= n; i++) {
            dp[i][0] = 0;
            dp[i][1] = 1;
        }
        for (int i = 0; i <= k; i++) {
```

```
            dp[1][i] = i;
        }
        for (int floor = 2; floor <= k; floor++) {
            for (int egg = 2; egg <= n; egg++) {
                int mini = Integer.MAX_VALUE;
                for (int x = 1; x <= floor; x++) {
                    int maxTrials = Math.max(dp[egg - 1][floor - x], dp[egg][x - 1]);
                    mini = Math.min(mini, maxTrials);
                }
                dp[egg][floor] = mini + 1;
            }
        }
        return dp[n][k];
    }

    static int f(int egg, int floor, int dp[][]) {
        if (floor == 1 || floor == 0) {
            return floor;
        }
        if (egg == 1) {
            return floor;
        }
        if (dp[egg][floor] != -1) {
            return dp[egg][floor];
        }
        int mini = Integer.MAX_VALUE;
        for (int x = 1; x <= floor; x++) {
            int maxTrials = Math.max(f(egg - 1, floor - x, dp), f(egg, x - 1, dp));
            mini = Math.min(mini, maxTrials);
        }
        return dp[egg][floor] = mini + 1;
    }
}

// Maximum Length Chain of Pairs
class Solution {
    int maxChainLength(Pair arr[], int n) {
        // your code here
        int dp[][] = new int[n + 1][n + 1];
        for (int rows[] : dp) {
            Arrays.fill(rows, -1);
        }

        Arrays.sort(arr, (g, y) -> (Integer.compare(g.x, y.x)));
        return f(0, -1, arr, dp, n);

    }

    int f(int ind, int prev, Pair arr[], int dp[][], int n) {
```

```
        if (ind == n) {
            return 0;
        }
        if (dp[ind][prev + 1] != -1) {
            return dp[ind][prev + 1];
        }
        int notTake = f(ind + 1, prev, arr, dp, n);
        int take = 0;
        if (prev == -1 || arr[ind].x > arr[prev].y) {
            take = 1 + f(ind + 1, ind, arr, dp, n);
        }
        return dp[ind][prev + 1] = Math.max(take, notTake);

    }
}

// Maximum size square sub-matrix with all 1s
class Solution {
    static int maxSquare(int n, int m, int mat[][]) {
        // code here
        int dp[][] = new int[n][m];
        int ans = 0;
        for (int i = 0; i < n; i++) {
            dp[i][0] = mat[i][0];
        }
        for (int i = 0; i < m; i++) {
            dp[0][i] = mat[0][i];
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (i == 0 || j == 0) {
                    ans = Math.max(dp[i][j], ans);
                    continue;
                }
                if (mat[i][j] == 1) {
                    dp[i][j] = 1 + Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1]));
                } else
                    dp[i][j] = 0;
                ans = Math.max(dp[i][j], ans);
            }
        }
        return ans;
    }
}

// Maximum sum of pairs with specific difference
// Min Cost Path Problem
class Solution {
    static int maximumPath(int N, int Matrix[][]) {
```

```
        // code here
        int dp[][] = new int[N][N];
        for (int row[] : dp) {
            Arrays.fill(row, -1);
        }
        int maxi = 0;
        for (int i = 0; i < N; i++) {
            maxi = Math.max(maxi, f(N - 1, i, N, Matrix, dp));
        }
        return maxi;
    }

    static int f(int row, int col, int N, int mat[][], int dp[][]) {
        if (col < 0 || col > N - 1) {
            return -(int) (1e9);
        }
        if (row == 0) {
            return mat[0][col];
        }

        if (dp[row][col] != -1) {
            return dp[row][col];
        }
        int up = mat[row][col] + f(row - 1, col, N, mat, dp);
        int upR = mat[row][col] + f(row - 1, col + 1, N, mat, dp);
        int upL = mat[row][col] + f(row - 1, col - 1, N, mat, dp);
        return dp[row][col] = Math.max(up, Math.max(upR, upL));
    }
}

// Maximum difference of zeros and ones in binary string
class Solution {
    int maxSubstring(String s) {
        // code here
        int n = s.length();
        if (allones(s, n)) {
            return -1;
        }
        int arr[] = new int[100000];
        for (int i = 0; i < n; i++) {
            arr[i] = (s.charAt(i) == '0' ? 1 : -1);
        }
        int dp[][] = new int[100000][3];
        for (int[] row : dp) {
            Arrays.fill(row, -1);
        }
        return findlength(arr, s, n, 0, 0, dp);
    }
```

```
    boolean allones(String s, int n) {
        int co = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '1') {
                co += 1;
            }
        }
        return (co == n);
    }

    int findlength(int arr[], String s, int n, int ind, int st, int dp[][]) {
        if (ind >= n) {
            return 0;
        }
        if (dp[ind][st] != -1) {
            return dp[ind][st];
        }

        if (st == 0) {
            return dp[ind][st] = Math.max(arr[ind] + findlength(arr, s, n, ind + 1, 1, dp),
                    0 + findlength(arr, s, n, ind + 1, 0, dp));
        }

        else {
            return dp[ind][st] = Math.max(arr[ind] + findlength(arr, s, n, ind + 1, 1, dp),
0);
        }
    }

    int maxSubstring(String s) {
        // code here
        int n = s.length();
        int arr[] = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = (s.charAt(i) == '0' ? 1 : -1);
        }
        int max_so_far = Integer.MIN_VALUE;
        int max_ending_here = 0;
        for (int i = 0; i < n; i++) {
            max_ending_here += arr[i];
            if (max_ending_here > max_so_far) {
                max_so_far = max_ending_here;
            }
            if (max_ending_here < 0) {
                max_ending_here = 0;
            }
        }
        return max_so_far;
    }
```

```
}

// Minimum number of jumps to reach end
// Minimum cost to fill given weight in a bag
class Solution {
    public int minimumCost(int cost[], int N, int W) {
        // Your code goes here
        int dp[][] = new int[N][W + 1];
        for (int rows[] : dp) {
            Arrays.fill(rows, -1);
        }
        return f(N - 1, W, cost, dp, N);
    }

    public int f(int ind, int W, int cost[], int dp[][], int N) {
        if (W < 0) {
            return (int) (1e9);
        }
        if (W == 0) {
            return 0;
        }
        if (ind == 0) {
            return cost[ind] == -1 ? (int) (1e9) : W * cost[0];
        }
        if (dp[ind][W] != -1) {
            return dp[ind][W];
        }
        int notTake = f(ind - 1, W, cost, dp, N);
        int take = Integer.MAX_VALUE;
        if (cost[ind] != -1) {
            take = cost[ind] + f(ind, W - (ind + 1), cost, dp, N);
        }
        return dp[ind][W] = Math.min(take, notTake);
    }
}

// Minimum removals from array to make max –min <= K
class Solution {
    int removals(int[] arr, int n, int k) {
        // code here
        Arrays.sort(arr);
        return f(0, n - 1, arr, dp, k);

    }

    int f(int s, int e, int arr[], int dp[][], int k) {
        if (s == e) {
            return 0;
        }
```

```
        if (dp[s][e] != -1) {
            return dp[s][e];
        }
        if (arr[e] - arr[s] <= k) {
            return 0;
        }
        int one = f(s + 1, e, arr, dp, k);
        int two = f(s, e - 1, arr, dp, k);
        return dp[s][e] = 1 + Math.min(one, two);
    }

}

// Longest Common Substring
class Solution {
    int longestCommonSubstr(String s1, String s2, int x, int y) {
        // code here
        int dp[][] = new int[x + 1][y + 1];
        for (int rows[] : dp) {
            Arrays.fill(rows, -1);
        }
        for (int i = 0; i <= x; i++) {
            dp[i][0] = 0;
        }
        for (int i = 0; i <= y; i++) {
            dp[0][i] = 0;
        }
        int ans = 0;
        for (int ind1 = 1; ind1 <= x; ind1++) {
            for (int ind2 = 1; ind2 <= y; ind2++) {
                if (s1.charAt(ind1 - 1) == s2.charAt(ind2 - 1)) {
                    dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
                    ans = Math.max(ans, dp[ind1][ind2]);
                } else {
                    dp[ind1][ind2] = 0;
                }
            }
        }
        return ans;
    }
}

// Count number of ways to reach a given score in a game
class Solution {
    public long count(int n) {
        long[] dp = new long[(int) n + 1];
        Arrays.fill(dp, 0);
        dp[0] = 1;
```

```
        // Add your code here.
        for (int i = 3; i <= n; i++) {
            dp[i] += dp[i - 3];
        }
        for (int i = 5; i <= n; i++) {
            dp[i] += dp[i - 5];
        }
        for (int i = 10; i <= n; i++) {
            dp[i] += dp[i - 10];
        }
        return dp[n];
    }
}

// Count Balanced Binary Trees of Height h
// LargestSum Contiguous Subarray [V>V>V>V IMP ]
class Solution {

    // arr: input array
    // n: size of array
    // Function to find the sum of contiguous subarray with maximum sum.
    long maxSubarraySum(int arr[], int n) {

        // Your code here
        long max_so_far = Integer.MIN_VALUE;
        long max_ending_here = 0;
        for (int i = 0; i < n; i++) {
            max_ending_here += arr[i];
            if (max_ending_here > max_so_far) {
                max_so_far = max_ending_here;
            }
            if (max_ending_here < 0) {
                max_ending_here = 0;
            }
        }
        return max_so_far;

    }

}

// Smallest sum contiguous subarray
class Solution {
    static int smallestSumSubarray(int arr[], int size) {
        // your code here
        int min_so_far = Integer.MAX_VALUE;
        int min_ending_here = 0;
        for (int i = 0; i < size; i++) {
            min_ending_here += arr[i];
```

```
            if (min_ending_here < min_so_far) {
                min_so_far = min_ending_here;
            }
            if (min_ending_here > 0) {
                min_ending_here = 0;
            }
        }
        return min_so_far;
    }
}

// Unbounded Knapsack (Repetition of items allowed)
class Solution {
    class TUF {
        static int knapsackUtil(int[] wt, int[] val, int ind, int W, int[][] dp) {
            if (ind == 0) {
                return ((int) (W / wt[0])) * val[0];
            }
            if (dp[ind][W] != -1)
                return dp[ind][W];
            int notTaken = 0 + knapsackUtil(wt, val, ind - 1, W, dp);
            int taken = Integer.MIN_VALUE;
            if (wt[ind] <= W)
                taken = val[ind] + knapsackUtil(wt, val, ind, W - wt[ind], dp);
            return dp[ind][W] = Math.max(notTaken, taken);
        }
    }

    class TUF {
        static int unboundedKnapsack(int n, int W, int[] val, int[] wt) {
            int[][] dp = new int[n][W + 1];
            for (int i = wt[0]; i <= W; i++) {
                dp[0][i] = ((int) i / wt[0]) * val[0];
            }
            for (int ind = 1; ind < n; ind++) {
                for (int cap = 0; cap <= W; cap++) {
                    int notTaken = 0 + dp[ind - 1][cap];
                    int taken = Integer.MIN_VALUE;
                    if (wt[ind] <= cap)
                        taken = val[ind] + dp[ind][cap - wt[ind]];
                    dp[ind][cap] = Math.max(notTaken, taken);
                }
            }
            return dp[n - 1][W];
        }
    }
}

// Word Break Problem
```

```
// Largest Independent Set Problem
// Partition problem
class Solution {
    class TUF {
        static boolean subsetSumUtil(int ind, int target, int arr[], int[][] dp) {
            if (target == 0)
                return true;
            if (ind == 0)
                return arr[0] == target;
            if (dp[ind][target] != -1)
                return dp[ind][target] == 0 ? false : true;
            boolean notTaken = subsetSumUtil(ind - 1, target, arr, dp);
            boolean taken = false;
            if (arr[ind] <= target)
                taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);
            dp[ind][target] = notTaken || taken ? 1 : 0;
            return notTaken || taken;
        }

        static boolean canPartition(int n, int[] arr) {
            int totSum = 0;
            for (int i = 0; i < n; i++) {
                totSum += arr[i];
            }
            if (totSum % 2 == 1)
                return false;
            else {
                int k = totSum / 2;
                int dp[][] = new int[n][k + 1];
                for (int row[] : dp)
                    Arrays.fill(row, -1);
                return subsetSumUtil(n - 1, k, arr, dp);
            }
        }
    }

    class TUF {
        static boolean canPartition(int n, int[] arr) {
            int totSum = 0;
            for (int i = 0; i < n; i++) {
                totSum += arr[i];
            }
            if (totSum % 2 == 1)
                return false;
            else {
                int k = totSum / 2;
                boolean dp[][] = new boolean[n][k + 1];
                for (int i = 0; i < n; i++) {
                    dp[i][0] = true;
```

```
            }
            if (arr[0] <= k)
                dp[0][arr[0]] = true;
            for (int ind = 1; ind < n; ind++) {
                for (int target = 1; target <= k; target++) {
                    boolean notTaken = dp[ind - 1][target];
                    boolean taken = false;
                    if (arr[ind] <= target)
                        taken = dp[ind - 1][target - arr[ind]];
                    dp[ind][target] = notTaken || taken;
                }
            }
            return dp[n - 1][k];
        }
    }
}
// Longest Palindromic Subsequence
class Solution {
    class TUF {
        static int lcs(String s1, String s2) {
            int n = s1.length();
            int m = s2.length();
            int dp[][] = new int[n + 1][m + 1];
            for (int rows[] : dp)
                Arrays.fill(rows, -1);
            for (int i = 0; i <= n; i++) {
                dp[i][0] = 0;
            }
            for (int i = 0; i <= m; i++) {
                dp[0][i] = 0;
            }
            for (int ind1 = 1; ind1 <= n; ind1++) {
                for (int ind2 = 1; ind2 <= m; ind2++) {
                    if (s1.charAt(ind1 - 1) == s2.charAt(ind2 - 1))
                        dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];
                    else
                        dp[ind1][ind2] = 0 + Math.max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);
                }
            }
            return dp[n][m];
        }

        static int longestPalindromeSubsequence(String s) {
            String t = s;
            String ss = new StringBuilder(s).reverse().toString();
            return lcs(ss, t);
        }
```

```
    }
}
// Count All Palindromic Subsequence in a given String
class Solution {
    long countPS(String str) {
        // Your code here
        int s = 0, e = str.length();
        long dp[][] = new long[e + 1][e + 1];
        for (long row[] : dp) {
            Arrays.fill(row, -1);
        }
        return f(s, e - 1, str, dp);
    }

    long f(int start, int end, String str, long dp[][]) {
        if (start == end) {
            return 1;
        }
        if (start > end) {
            return 0;
        }
        if (dp[start][end] != -1) {
            return dp[start][end];
        }
        if (str.charAt(start) == str.charAt(end)) {
            return dp[start][end] = 1 + f(start + 1, end, str, dp) + f(start, end - 1, str, dp);
        } else
            return dp[start][end] = f(start + 1, end, str, dp) + f(start, end - 1, str, dp)
                    - f(start + 1, end - 1, str, dp);
    }
}

// Longest Palindromic Substring
class Solution {
    static String longestPalin(String S) {
        // code here
        int l1 = S.length();
        int l, h, start = 0, end = 1;
        for (int i = 1; i < l1; i++) {
            // Even Palindrome
            l = i - 1;
            h = i;
            while (l >= 0 && h < l1 && S.charAt(l) == S.charAt(h)) {
                if (h - l + 1 > end) {
                    start = l;
                    end = h - l + 1;
                }
                l--;
                h++;
```

```
            }
            // Odd Substring
            l = i - 1;
            h = i + 1;

            while (l >= 0 && h < l1 && S.charAt(l) == S.charAt(h)) {
                if (h - l + 1 > end) {
                    start = l;
                    end = h - l + 1;
                }
                l--;
                h++;
            }
        }
        return S.substring(start, start + end);
    }
}
// Longest alternating subsequence
class Solution {
    public int AlternatingaMaxLength(int[] nums) {
        // code here
        int n = nums.length;
        int dp[][][] = new int[n + 1][n + 1][2];
        for (int rows[][] : dp) {
            for (int y[] : rows) {
                Arrays.fill(y, -1);
            }
        }
        return Math.max(f(0, -1, 0, nums, dp), f(0, -1, 1, nums, dp));
    }

    public int f(int ind, int prev, int gret, int nums[], int dp[][][]) {
        if (ind == nums.length) {
            return 0;
        }
        if (dp[ind][prev + 1][gret] != -1) {
            return dp[ind][prev + 1][gret];
        }
        if (gret == 1) {
            if (prev == -1 || nums[ind] > nums[prev]) {
                return dp[ind][prev + 1][gret] = Math.max(1 + f(ind + 1, ind, 0, nums, dp),
                        f(ind + 1, prev, 1, nums, dp));
            } else {
                return dp[ind][prev + 1][gret] = f(ind + 1, prev, 1, nums, dp);
            }
        } else {
            if (prev == -1 || nums[ind] < nums[prev]) {
                return dp[ind][prev + 1][gret] = Math.max(1 + f(ind + 1, ind, 1, nums, dp),
                        f(ind + 1, prev, 0, nums, dp));
```

```
            } else {
                return dp[ind][prev + 1][gret] = f(ind + 1, prev, 0, nums, dp);
            }
        }

    }
}

// Weighted Job Scheduling
// Coin game winner where every player has three choices
// Count Derangements (Permutation such that no element appears in its original
// position) [ IMPORTANT ]
// Maximum profit by buying and selling a share at most twice [ IMP ]
class Solution {
    static int maxProfit(int K, int N, int A[]) {
        // code here
        int dp[][][] = new int[N + 1][2][K + 1];
        for (int lp[][] : dp) {
            for (int r[] : lp) {
                Arrays.fill(r, -1);
            }
        }
        return f(0, 0, K, dp, N, A);
    }

    static int f(int ind, int buy, int cap, int dp[][][], int n, int A[]) {
        if (ind == n) {
            return 0;
        }
        if (cap == 0) {
            return 0;
        }
        if (dp[ind][buy][cap] != -1) {
            return dp[ind][buy][cap];
        }
        if (buy == 0) {
            return dp[ind][buy][cap] = Math.max(f(ind + 1, 0, cap, dp, n, A),
                    -A[ind] + f(ind + 1, 1, cap, dp, n, A));
        } else
            return dp[ind][buy][cap] = Math.max(f(ind + 1, 1, cap, dp, n, A),
                    A[ind] + f(ind + 1, 0, cap - 1, dp, n, A));

    }
}

// Optimal Strategy for a Game
class Solution {
    // Function to find the maximum possible amount of money we can win.
    static long countMaximum(int arr[], int n) {
```

```
            // Your code here
            int dp[][] = new int[n][n];
            for (int rows[] : dp) {
                Arrays.fill(rows, -1);
            }
            return f(0, n - 1, arr, dp);
        }

        static int f(int start, int end, int arr[], int dp[][]) {
            if (start > end || start < 0 || end > arr.length - 1) {
                return 0;
            }
            if (dp[start][end] != -1) {
                return dp[start][end];
            }
            int pickFirst = arr[start] + Math.min(f(start + 2, end, arr, dp), f(start + 1, end - 1,
 arr, dp));
            int pickLast = arr[end] + Math.min(f(start + 1, end - 1, arr, dp), f(start, end - 2,
arr, dp));
            return dp[start][end] = Math.max(pickFirst, pickLast);
        }
    }

    // Optimal Binary Search Tree
    // Palindrome Partitioning Problem
    class Solution {
        int f(int i, String str) {
            if (i == str.length())
                return 0;
            if (dp[i] != -1)
                return dp[i];
            String temp = "";
            int minCost = Integer.MAX_VALUE;
            for (int j = i; j < str.length(); j++) {
                temp = temp + str.charAt(j);
                if (isPalindrome(temp) == true) {
                    int cost = 1 + f(j + 1, str);
                }
                minCost = Math.min(minCost, cost);
            }

            return dp[i] = minCost;
        }

        int f(int i, String str) {
            int dp[] = new int[n + 1];
            for (int i = 1; i < n; i++) {
                dp[i] = 0;
            }
```

```
        int n = str.length();
        for (int i = n - 1; i >= 1; i--) {
            int minCost = Integer.MAX_VALUE;
            for (int j = i; j < n; j++) {
                if (isPalindrome(i, j, str) == true) {
                    int cost = 1 + dp[j + 1];
                    minCost = Math.min(minCost, cost);
                }
            }
            dp[i] = minCost;
        }
        return dp[0] - 1;
    }
}

// Word Wrap Problem
// Mobile Numeric Keypad Problem [ IMP ]
class Solution {
    public long getCount(int N) {
        // Your code goes here
        int R = 4, C = 3;
        int mat[][] = new int[][] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { -1, 0, -1 } };
        long dp[][] = new long[10][N + 1];
        for (long rows[] : dp) {
            Arrays.fill(rows, -1);
        }
        int dx[] = new int[] { 0, 1, 0, -1, 0 };
        int dy[] = new int[] { -1, 0, 1, 0, 0 };
        long cnt = 0;
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 3; j++) {
                if (mat[i][j] >= 0) {
                    cnt += f(i, j, mat, dx, dy, 1, N, R, C, dp);
                }
            }
        }
        return cnt;
    }

    public long f(int row, int col, int mat[][], int[] dx, int[] dy, int c, int N, int R, int C,
            long dp[][]) {
        // System.out.println("Row: "+row+" Col: "+col+" Count:"+c);
        if (c == N) {
            return 1;
        }
        if (dp[mat[row][col]][c] != -1) {
            return dp[mat[row][col]][c];
        }
        long count = 0;
```

```
            for (int i = 0; i < 5; i++) {
                int nr = row + dx[i];
                int nc = col + dy[i];
                if (nr >= 0 && nr < R && nc >= 0 && nc < C && mat[nr][nc] >= 0) {
                    count += f(nr, nc, mat, dx, dy, c + 1, N, R, C, dp);
                }
            }
            return dp[mat[row][col]][c] = count;
        }

    }

    // Boolean Parenthesization Problem
    class Solution {
        int f(int i, int j, boolean isTrue, String str) {
            if (i > j) return 0;
            if (i == j) {
            if (isTrue == 1) {
                return str.charAt(i) == 'T';
            } else return str.charAt(i) == 'F';
            }
            if (dp[i][j][isTrue] != -1) return dp[i][j][isTrue];
            int ways = 0;
            for (int ind = i + 1; ind <= j - 1; ind = ind + 2) {
                int LeftTrue = f(i, ind - 1, 1, str);
                int LeftFalse = f(i, ind - 1, 0, str);
                int RightTrue = f(ind + 1, j, 1, str);
                int RightFalse = f(ind + 1, j, 0, str);

                if (str.charAt(ind) == '&') {
                    if (isTrue == true) ways = ways + (LeftTrue * RightTrue);
                    else ways = ways + (LeftFalse * RightTrue) + (LeftTrue * RightFalse) +
(LeftFalse * RightFalse);
                } else if (str.charAt(ind) == '|') {
                    if (isTrue == true) ways = ways + (LeftFalse * RightTrue) + (LeftTrue *
RightFalse) + (LeftTrue * RightTrue);
                    else ways = ways + (LeftFalse * RightFalse);
                } else if (str.charAt(ind) == '^') {
                    if (isTrue == true) ways = ways + ((LeftFalse * RightTrue) + (LeftTrue *
RightFalse))
                    else ways = ways + (LeftTrue * RightTrue) + (LeftFalse * RightFalse);
                }
            }
            return dp[i][j][isTrue]=ways;
            }
    }

    // Largest rectangular sub-matrix whose sum is 0
    class Solution {
```

```
class Solution {
    public static ArrayList<ArrayList<Integer>> sumZeroMatrix(int[][] M) {
        // code here
        int R = M.length;
        int C = M[0].length;
        int max_area = 0, start_col = 0, end_col = 0, start_row = 0, end_row = 0;
        for (int slab = 0; slab < R; slab++) {
            int ColPrefix[] = new int[C];
            for (int moving = slab; moving < R; moving++) {
                for (int col = 0; col < C; col++) {
                    ColPrefix[col] += M[moving][col];
                }

                int len[] = f(ColPrefix, C);
                int area = (moving - slab + 1) * len[2];

                if (area > max_area) {
                    start_row = slab;
                    end_row = moving;
                    start_col = len[0];
                    end_col = len[1];
                    max_area = area;
                }
            }
        }
        ArrayList<ArrayList<Integer>> ans = new ArrayList<>();
        if (start_row == 0 && start_col == 0 && end_row == 0 && end_col == 0) {
            return ans;
        }
        for (int i = start_row; i <= end_row; i++) {
            ArrayList<Integer> ls = new ArrayList<>();
            for (int j = start_col; j <= end_col; j++) {
                ls.add(M[i][j]);
            }
            ans.add(ls);
        }
        return ans;
    }

    public static int[] f(int arr[], int n) {
        Map<Integer, Integer> mp = new HashMap<>();
        int max_len = 0;
        int ans[] = new int[3];
        int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += arr[i];
            if (arr[i] == 0 && max_len == 0) {
                max_len = 1;
                ans[0] = i;
```

```
                ans[1] = i;
                ans[2] = 1;
            }
            if (sum == 0 && max_len < i + 1) {
                max_len = i + 1;
                ans[0] = 0;
                ans[1] = i;
                ans[2] = i + 1;
            }
            if (mp.containsKey(sum)) {
                if (i - mp.get(sum) >= max_len) {

                    ans[0] = mp.get(sum) == -1 ? 0 : mp.get(sum);
                    ans[0]++;
                    ans[1] = i;
                    ans[2] = ans[1] - ans[0];
                }
            } else
                mp.put(sum, i);
        }

        return ans;
    }

}

public static ArrayList<ArrayList<Integer>> sumZeroMatrix(int[][] M) {
    // code here
    int R = M.length;
    int C = M[0].length;
    int max_area = 0, start_col = 0, end_col = 0, start_row = 0, end_row = 0;
    for (int slab = 0; slab < R; slab++) {
        int ColPrefix[] = new int[C];
        for (int moving = slab; moving < R; moving++) {
            for (int col = 0; col < C; col++) {
                ColPrefix[col] += M[moving][col];
            }
            int len[] = f(ColPrefix, C);
            int area = (moving - slab + 1) * (len[1] - len[0]);
            if (area > max_area) {
                start_row = slab;
                end_row = moving;
                start_col = len[0];
                end_col = len[1];
                max_area = area;
            }
        }
    }
    ArrayList<ArrayList<Integer>> ans = new ArrayList<>();
```

```
        for (int i = start_row; i <= end_row; i++) {
            ArrayList<Integer> ls = new ArrayList<>();
            for (int j = start_col; j <= end_col; j++) {
                ls.add(M[i][j]);
            }
            ans.add(ls);
        }
        return ans;
    }

    public static int[] f(int arr[], int n) {
        Map<Integer, Integer> mp = new HashMap<>();
        int max_len = 0;
        int ans[] = new int[2];
        int sum = 0;
        mp.put(0, -1);
        for (int i = 0; i < n; i++) {
            sum += arr[i];
            if (mp.containsKey(sum)) {
                if (i - mp.get(sum) > max_len) {
                    ans[0] = mp.get(sum) == -1 ? 0 : mp.get(sum);
                    ans[1] = i;
                }
            } else
                mp.put(sum, i);
        }
        return ans;
    }
}

// Largest area rectangular sub-matrix with equal number of 1's and 0's [ IMP ]
// Maximum sum rectangle in a 2D matrix
class Solution {
    int Kadane(int arr[], int n) {
        int csum = 0, msum = Integer.MIN_VALUE;
        for (int i = 0; i < n; i++) {
            csum += arr[i];
            msum = Math.max(csum, msum);
            if (csum < 0) {
                csum = 0;
            }
        }
        return msum;
    }

    int maximumSumRectangle(int R, int C, int M[][]) {
        // code here
        int ans = Integer.MIN_VALUE;
        for (int slab = 0; slab < R; slab++) {
```

```
                int ColPrefix[] = new int[C];
                for (int moving = slab; moving < R; moving++) {
                    for (int col = 0; col < C; col++) {
                        ColPrefix[col] += M[moving][col];
                    }
                    ans = Math.max(ans, Kadane(ColPrefix, C));
                }
            }
            return ans;
        }
    };

    // Maximum profit by buying and selling a share at most k times
    class Solution {
        static int maxProfit(int K, int N, int A[]) {
            // code here
            int dp[][][] = new int[N + 1][2][K + 1];
            for (int lp[][] : dp) {
                for (int r[] : lp) {
                    Arrays.fill(r, -1);
                }
            }
            return f(0, 0, K, dp, N, A);
        }

        static int f(int ind, int buy, int cap, int dp[][][], int n, int A[]) {
            if (ind == n) {
                return 0;
            }
            if (cap == 0) {
                return 0;
            }
            if (dp[ind][buy][cap] != -1) {
                return dp[ind][buy][cap];
            }
            if (buy == 0) {
                return dp[ind][buy][cap] = Math.max(f(ind + 1, 0, cap, dp, n, A),
                        -A[ind] + f(ind + 1, 1, cap, dp, n, A));
            } else
                return dp[ind][buy][cap] = Math.max(f(ind + 1, 1, cap, dp, n, A),
                        A[ind] + f(ind + 1, 0, cap - 1, dp, n, A));

        }
    }
    // Find if a string is interleaved of two other strings
    // Maximum Length of Pair Chain

  }
}
```