/\* Dynamic Programming Java implementation of LCS problem \*/

public class LongestCommonSubsequence

{

  /\* Returns length of LCS for X[0..m-1], Y[0..n-1] \*/

  int lcs( char[] X, char[] Y, int m, int n )

  {

    int L[][] = new int[m+1][n+1];

    /\* Following steps build L[m+1][n+1] in bottom up fashion. Note

         that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] \*/

    for (int i=0; i<=m; i++)

    {

      for (int j=0; j<=n; j++)

      {

        if (i == 0 || j == 0)

            L[i][j] = 0;

        else if (X[i-1] == Y[j-1])

            L[i][j] = L[i-1][j-1] + 1;

        else

            L[i][j] = max(L[i-1][j], L[i][j-1]);

      }

    }

  return L[m][n];

  }

  /\* Utility function to get max of 2 integers \*/

  int max(int a, int b)

  {

    return (a > b)? a : b;

  }

  public static void main(String[] args)

  {

    LongestCommonSubsequence lcs = new LongestCommonSubsequence();

    String s1 = "AGGTAB";

    String s2 = "GXTXAYB";

    char[] X=s1.toCharArray();

    char[] Y=s2.toCharArray();

    int m = X.length;

    int n = Y.length;

    System.out.println("Length of LCS is" + " " +

                                  lcs.lcs( X, Y, m, n ) );

  }

}

/\* Dynamic Programming Java implementation of LIS problem \*/

class LIS

{

    /\* lis() returns the length of the longest increasing

       subsequence in arr[] of size n \*/

    static int lis(int arr[],int n)

    {

          int lis[] = new int[n];

          int i,j,max = 0;

          /\* Initialize LIS values for all indexes \*/

           for ( i = 0; i < n; i++ )

              lis[i] = 1;

           /\* Compute optimized LIS values in bottom up manner \*/

           for ( i = 1; i < n; i++ )

              for ( j = 0; j < i; j++ )

                         if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)

                    lis[i] = lis[j] + 1;

           /\* Pick maximum of all LIS values \*/

           for ( i = 0; i < n; i++ )

              if ( max < lis[i] )

                 max = lis[i];

            return max;

    }

    public static void main(String args[])

    {

        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };

            int n = arr.length;

            System.out.println("Length of lis is " + lis( arr, n ) + "n" );

    }

}

// A Dynamic Programming based Java program to find minimum

// number operations to convert str1 to str2

class EDIST

{

    static int min(int x,int y,int z)

    {

        if (x <= y && x <= z) return x;

        if (y <= x && y <= z) return y;

        else return z;

    }

    static int editDistDP(String str1, String str2, int m, int n)

    {

        // Create a table to store results of subproblems

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

        // Fill d[][] in bottom up manner

        for (int i=0; i<=m; i++)

        {

            for (int j=0; j<=n; j++)

            {

                // If first string is empty, only option is to

                // isnert all characters of second string

                if (i==0)

                    dp[i][j] = j;  // Min. operations = j

                // If second string is empty, only option is to

                // remove all characters of second string

                else if (j==0)

                    dp[i][j] = i; // Min. operations = i

                // If last characters are same, ignore last char

                // and recur for remaining string

                else if (str1.charAt(i-1) == str2.charAt(j-1))

                    dp[i][j] = dp[i-1][j-1];

                // If the last character is different, consider all

                // possibilities and find the minimum

                else

                    dp[i][j] = 1 + min(dp[i][j-1],  // Insert

                                       dp[i-1][j],  // Remove

                                       dp[i-1][j-1]); // Replace

            }

        }

        return dp[m][n];

    }

    public static void main(String args[])

    {

        String str1 = "sunday";

        String str2 = "saturday";

        System.out.println( editDistDP( str1 , str2 , str1.length(), str2.length()) );

    }

}/\*This code is contributed by Rajat Mishra\*/

// A Dynamic Programming based Java program to find minimum

// number operations to convert str1 to str2

class EDIST

{

    static int min(int x,int y,int z)

    {

        if (x <= y && x <= z) return x;

        if (y <= x && y <= z) return y;

        else return z;

    }

    static int editDistDP(String str1, String str2, int m, int n)

    {

        // Create a table to store results of subproblems

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

        // Fill d[][] in bottom up manner

        for (int i=0; i<=m; i++)

        {

            for (int j=0; j<=n; j++)

            {

                // If first string is empty, only option is to

                // isnert all characters of second string

                if (i==0)

                    dp[i][j] = j;  // Min. operations = j

                // If second string is empty, only option is to

                // remove all characters of second string

                else if (j==0)

                    dp[i][j] = i; // Min. operations = i

                // If last characters are same, ignore last char

                // and recur for remaining string

                else if (str1.charAt(i-1) == str2.charAt(j-1))

                    dp[i][j] = dp[i-1][j-1];

                // If the last character is different, consider all

                // possibilities and find the minimum

                else

                    dp[i][j] = 1 + min(dp[i][j-1],  // Insert

                                       dp[i-1][j],  // Remove

                                       dp[i-1][j-1]); // Replace

            }

        }

        return dp[m][n];

    }

    public static void main(String args[])

    {

        String str1 = "sunday";

        String str2 = "saturday";

        System.out.println( editDistDP( str1 , str2 , str1.length(), str2.length()) );

    }

}/\*This code is contributed by Rajat Mishra\*/

/ A Recursive java program to solve

// minimum sum partition problem.

import java.io.\*;

class GFG

{

    // Returns the minimum value of

    //the difference of the two sets.

    static int findMin(int arr[], int n)

    {

        // Calculate sum of all elements

        int sum = 0;

        for (int i = 0; i < n; i++)

            sum += arr[i];

        // Create an array to store

        // results of subproblems

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

        // Initialize first column as true.

        // 0 sum is possible  with all elements.

        for (int i = 0; i <= n; i++)

            dp[i][0] = true;

        // Initialize top row, except dp[0][0],

        // as false. With 0 elements, no other

        // sum except 0 is possible

        for (int i = 1; i <= sum; i++)

            dp[0][i] = false;

        // Fill the partition table

        // in bottom up manner

        for (int i = 1; i <= n; i++)

        {

            for (int j = 1; j <= sum; j++)

            {

                // If i'th element is excluded

                dp[i][j] = dp[i - 1][j];

                // If i'th element is included

                if (arr[i - 1] <= j)

                    dp[i][j] |= dp[i - 1][j - arr[i - 1]];

            }

        }

        // Initialize difference of two sums.

        int diff = Integer.MAX\_VALUE;

        // Find the largest j such that dp[n][j]

        // is true where j loops from sum/2 t0 0

        for (int j = sum / 2; j >= 0; j--)

        {

            // Find the

            if (dp[n][j] == true)

            {

                diff = sum - 2 \* j;

                break;

            }

        }

        return diff;

    }

    // Driver program

    public static void main (String[] args)

    {

        int arr[] = {3, 1, 4, 2, 2, 1};

        int n = arr.length;

        System.out.println ("The minimum difference between 2 sets is "

                            + findMin(arr, n));

    }

}

// This code is contributed by vt\_m

// A Dynamic Programming based Java program

// to count number of ways to cover a distance

// with 1, 2 and 3 steps

import java.io.\*;

class GFG

{

    // Function returns count of ways to cover 'dist'

    static int printCountDP(int dist)

    {

        int[] count = new int[dist+1];

        // Initialize base values. There is one way to

        // cover 0 and 1 distances and two ways to

        // cover 2 distance

        count[0] = 1;

        count[1] = 1;

        count[2] = 2;

        // Fill the count array in bottom up manner

        for (int i=3; i<=dist; i++)

            count[i] = count[i-1] + count[i-2] + count[i-3];

        return count[dist];

    }

    // driver program

    public static void main (String[] args)

    {

        int dist = 4;

        System.out.println(printCountDP(dist));

    }

}

// This code is contributed by Pramod Kumar

// Java program to find the longest path in a matrix

// with given constraints

class GFG

{

    public static int n = 3;

    // Function that returns length of the longest path

    // beginning with mat[i][j]

    // This function mainly uses lookup table dp[n][n]

    static int findLongestFromACell(int i, int j, int mat[][], int dp[][])

    {

        // Base case

        if (i<0 || i>=n || j<0 || j>=n)

            return 0;

        // If this subproblem is already solved

        if (dp[i][j] != -1)

            return dp[i][j];

        // Since all numbers are unique and in range from 1 to n\*n,

        // there is atmost one possible direction from any cell

        if (j<n-1 && ((mat[i][j] +1) == mat[i][j+1]))

            return dp[i][j] = 1 + findLongestFromACell(i,j+1,mat,dp);

        if (j>0 && (mat[i][j] +1 == mat[i][j-1]))

            return dp[i][j] = 1 + findLongestFromACell(i,j-1,mat,dp);

        if (i>0 && (mat[i][j] +1 == mat[i-1][j]))

            return dp[i][j] = 1 + findLongestFromACell(i-1,j,mat,dp);

        if (i<n-1 && (mat[i][j] +1 == mat[i+1][j]))

            return dp[i][j] = 1 + findLongestFromACell(i+1,j,mat,dp);

        // If none of the adjacent fours is one greater

        return dp[i][j] = 1;

    }

    // Function that returns length of the longest path

    // beginning with any cell

    static int finLongestOverAll(int mat[][])

    {

        // Initialize result

        int result = 1;

        // Create a lookup table and fill all entries in it as -1

        int[][] dp = new int[n][n];

        for(int i=0;i<n;i++)

            for(int j=0;j<n;j++)

                dp[i][j] = -1;

        // Compute longest path beginning from all cells

        for (int i=0; i<n; i++)

        {

            for (int j=0; j<n; j++)

            {

                if (dp[i][j] == -1)

                    findLongestFromACell(i, j, mat, dp);

                //  Update result if needed

                result = Math.max(result, dp[i][j]);

            }

        }

        return result;

    }

    // driver program

    public static void main (String[] args)

    {

        int  mat[][] = { {1, 2, 9},

                         {5, 3, 8},

                         {4, 6, 7} };

        System.out.println("Length of the longest path is " +

                            finLongestOverAll(mat));

    }

}

// Contributed by Pramod Kumar

// A Dynamic Programming solution for subset

// sum problem

class GFG {

    // Returns true if there is a subset of

    // set[] with sun equal to given sum

    static boolean isSubsetSum(int set[],

                             int n, int sum)

    {

        // The value of subset[i][j] will be

        // true if there is a subset of

        // set[0..j-1] with sum equal to i

        boolean subset[][] =

                     new boolean[sum+1][n+1];

        // If sum is 0, then answer is true

        for (int i = 0; i <= n; i++)

            subset[0][i] = true;

        // If sum is not 0 and set is empty,

        // then answer is false

        for (int i = 1; i <= sum; i++)

            subset[i][0] = false;

        // Fill the subset table in botton

        // up manner

        for (int i = 1; i <= sum; i++)

        {

            for (int j = 1; j <= n; j++)

            {

                subset[i][j] = subset[i][j-1];

                if (i >= set[j-1])

                subset[i][j] = subset[i][j] ||

                     subset[i - set[j-1]][j-1];

            }

        }

        /\* // uncomment this code to print table

        for (int i = 0; i <= sum; i++)

        {

        for (int j = 0; j <= n; j++)

            System.out.println (subset[i][j]);

        } \*/

        return subset[sum][n];

    }

    /\* Driver program to test above function \*/

    public static void main (String args[])

    {

        int set[] = {3, 34, 4, 12, 5, 2};

        int sum = 9;

        int n = set.length;

        if (isSubsetSum(set, n, sum) == true)

            System.out.println("Found a subset"

                          + " with given sum");

        else

            System.out.println("No subset with"

                               + " given sum");

    }

}

/\* This code is contributed by Rajat Mishra \*/

(Juego en el que participan dos personas: hay una pila de monedas y cada jugador puede sacar la última o la primera moneda).

// Java program to find out maximum

// value from a given sequence of coins

import java.io.\*;

class GFG

{

    // Utility functions to get maximum

    // and minimum of two intgers

    int max(int a, int b) { return a > b ? a : b; }

    int min(int a, int b) { return a < b ? a : b; }

    // Returns optimal value possible that a player

    // can collect from an array of coins of size n.

    // Note than n must be even

    static int optimalStrategyOfGame(int arr[], int n)

    {

        // Create a table to store solutions of subproblems

        int table[][] = new int[n][n];

        int gap, i, j, x, y, z;

        // Fill table using above recursive formula.

        // Note that the tableis filled in diagonal

        // fashion (similar to <http://goo.gl/PQqoS>),

        // from diagonal elements to table[0][n-1]

        // which is the result.

        for (gap = 0; gap < n; ++gap)

        {

            for (i = 0, j = gap; j < n; ++i, ++j)

            {

                // Here x is value of F(i+2, j),

                //  y is F(i+1, j-1) and z is

                // F(i, j-2) in above recursive formula

                x = ((i + 2) <= j) ? table[i + 2][j] : 0;

                y = ((i + 1) <= (j - 1)) ? table[i +1 ][j -  1] : 0;

                z = (i <= (j - 2)) ? table[i][j - 2]: 0;

                table[i][j] = Math.max(arr[i] +

                              Math.min(x, y), arr[j] +

                              Math.min(y, z));

            }

        }

        return table[0][n - 1];

    }

    // Driver program

    public static void main (String[] args)

    {

        int arr1[] = {8, 15, 3, 7};

        int n = arr1.length;

        System.out.println("" + optimalStrategyOfGame(arr1, n));

        int arr2[] = {2, 2, 2, 2};

        n = arr2.length;

        System.out.println("" + optimalStrategyOfGame(arr2, n));

        int arr3[] = {20, 30, 2, 2, 2, 10};

        n = arr3.length;

        System.out.println("" + optimalStrategyOfGame(arr3, n));

    }

}

// This code is contributed by vt\_m

(Se dan número y peso de n items + peso máximo w de bolsa. ¿Valor m?)

/ A Dynamic Programming based solution for 0-1 Knapsack problem

class Knapsack

{

    // A utility function that returns maximum of two integers

    static int max(int a, int b) { return (a > b)? a : b; }

   // Returns the maximum value that can be put in a knapsack of capacity W

    static int knapSack(int W, int wt[], int val[], int n)

    {

         int i, w;

     int K[][] = new int[n+1][W+1];

     // Build table K[][] in bottom up manner

     for (i = 0; i <= n; i++)

     {

         for (w = 0; w <= W; w++)

         {

             if (i==0 || w==0)

                  K[i][w] = 0;

             else if (wt[i-1] <= w)

                   K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);

             else

                   K[i][w] = K[i-1][w];

         }

      }

      return K[n][W];

    }

    // Driver program to test above function

    public static void main(String args[])

    {

        int val[] = new int[]{60, 100, 120};

    int wt[] = new int[]{10, 20, 30};

    int  W = 50;

    int n = val.length;

    System.out.println(knapSack(W, wt, val, n));

    }

}

/\*This code is contributed by Rajat Mishra \*/

(Está en c++)

#include<iostream>

#include<cstring>

using namespace std;

// Returns count of all possible parenthesizations that lead to

// result true for a boolean expression with symbols like true

// and false and operators like &, | and ^ filled between symbols

int countParenth(char symb[], char oper[], int n)

{

    int F[n][n], T[n][n];

    // Fill diaginal entries first

    // All diagonal entries in T[i][i] are 1 if symbol[i]

    // is T (true).  Similarly, all F[i][i] entries are 1 if

    // symbol[i] is F (False)

    for (int i = 0; i < n; i++)

    {

        F[i][i] = (symb[i] == 'F')? 1: 0;

        T[i][i] = (symb[i] == 'T')? 1: 0;

    }

    // Now fill T[i][i+1], T[i][i+2], T[i][i+3]... in order

    // And F[i][i+1], F[i][i+2], F[i][i+3]... in order

    for (int gap=1; gap<n; ++gap)

    {

        for (int i=0, j=gap; j<n; ++i, ++j)

        {

            T[i][j] = F[i][j] = 0;

            for (int g=0; g<gap; g++)

            {

                // Find place of parenthesization using current value

                // of gap

                int k = i + g;

                // Store Total[i][k] and Total[k+1][j]

                int tik = T[i][k] + F[i][k];

                int tkj = T[k+1][j] + F[k+1][j];

                // Follow the recursive formulas according to the current

                // operator

                if (oper[k] == '&')

                {

                    T[i][j] += T[i][k]\*T[k+1][j];

                    F[i][j] += (tik\*tkj - T[i][k]\*T[k+1][j]);

                }

                if (oper[k] == '|')

                {

                    F[i][j] += F[i][k]\*F[k+1][j];

                    T[i][j] += (tik\*tkj - F[i][k]\*F[k+1][j]);

                }

                if (oper[k] == '^')

                {

                    T[i][j] += F[i][k]\*T[k+1][j] + T[i][k]\*F[k+1][j];

                    F[i][j] += T[i][k]\*T[k+1][j] + F[i][k]\*F[k+1][j];

                }

            }

        }

    }

    return T[0][n-1];

}

// Driver program to test above function

int main()

{

    char symbols[] = "TTFT";

    char operators[] = "|&^";

    int n = strlen(symbols);

    // There are 4 ways

    // ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and (T|((T&F)^T))

    cout << countParenth(symbols, operators, n);

    return 0;

}

(Dados dos strings, encontrar la cadena más corta tal que s1 y s2 sean subsecuencias de dicho string)

// A Naive recursive Java program to find

// length of the shortest supersequence

class GFG

{

    static int superSeq(String X, String Y,

                                  int m, int n)

    {

        if (m == 0) return n;

        if (n == 0) return m;

        if (X.charAt(m - 1) == Y.charAt(n - 1))

            return 1 + superSeq(X, Y, m - 1, n - 1);

        return 1 + Math.min(superSeq(X, Y, m - 1, n),

                    superSeq(X, Y, m, n - 1));

    }

    // Driver code

    public static void main(String args[])

    {

    String X = "AGGTAB";

    String Y = "GXTXAYB";

    System.out.println("Length of the shortest" +

                        "supersequence is: "

            + superSeq(X, Y, X.length(),Y.length()));

    }

}

// This article is contributed by Sumit Ghosh

(Given an array p[] which represents the chain of matrices such that the ith matrix Ai is of dimension p[i-1] x p[i]. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.)

// Dynamic Programming Python implementation of Matrix

// Chain Multiplication.

// See the Cormen book for details of the following algorithm

class MatrixChainMultiplication

{

    // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n

    static int MatrixChainOrder(int p[], int n)

    {

        /\* For simplicity of the program, one extra row and one

        extra column are allocated in m[][].  0th row and 0th

        column of m[][] are not used \*/

        int m[][] = new int[n][n];

        int i, j, k, L, q;

        /\* m[i,j] = Minimum number of scalar multiplications needed

        to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where

        dimension of A[i] is p[i-1] x p[i] \*/

        // cost is zero when multiplying one matrix.

        for (i = 1; i < n; i++)

            m[i][i] = 0;

        // L is chain length.

        for (L=2; L<n; L++)

        {

            for (i=1; i<n-L+1; i++)

            {

                j = i+L-1;

                if(j == n) continue;

                m[i][j] = Integer.MAX\_VALUE;

                for (k=i; k<=j-1; k++)

                {

                    // q = cost/scalar multiplications

                    q = m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j];

                    if (q < m[i][j])

                        m[i][j] = q;

                }

            }

        }

        return m[1][n-1];

    }

    // Driver program to test above function

    public static void main(String args[])

    {

        int arr[] = new int[] {1, 2, 3, 4};

        int size = arr.length;

        System.out.println("Minimum number of multiplications is "+

                           MatrixChainOrder(arr, size));

    }

}

/\* This code is contributed by Rajat Mishra\*/

(Dado un arreglo, es posible partirlo en 2 de tal forma que sus subarreglos tengan misma suma?

// A dynamic programming based Java program for partition problem

import java.io.\*;

class Partition {

    // Returns true if arr[] can be partitioned in two subsets of

    // equal sum, otherwise false

    static boolean findPartition (int arr[], int n)

    {

        int sum = 0;

        int i, j;

        // Caculcate sun of all elements

        for (i = 0; i < n; i++)

            sum += arr[i];

        if (sum%2 != 0)

            return false;

        boolean part[][]=new boolean[sum/2+1][n+1];

        // initialize top row as true

        for (i = 0; i <= n; i++)

            part[0][i] = true;

        // initialize leftmost column, except part[0][0], as 0

        for (i = 1; i <= sum/2; i++)

            part[i][0] = false;

        // Fill the partition table in botton up manner

        for (i = 1; i <= sum/2; i++)

        {

            for (j = 1; j <= n; j++)

            {

                part[i][j] = part[i][j-1];

                if (i >= arr[j-1])

                    part[i][j] = part[i][j] ||

                                 part[i - arr[j-1]][j-1];

            }

        }

        /\* // uncomment this part to print table

        for (i = 0; i <= sum/2; i++)

        {

            for (j = 0; j <= n; j++)

                printf ("%4d", part[i][j]);

            printf("\n");

        } \*/

        return part[sum/2][n];

    }

    /\*Driver function to check for above function\*/

    public static void main (String[] args)

    {

        int arr[] = {3, 1, 1, 2, 2,1};

        int n = arr.length;

        if (findPartition(arr, n) == true)

            System.out.println("Can be divided into two "

                               "subsets of equal sum");

        else

            System.out.println("Can not be divided into"

                            " two subsets of equal sum");

    }

}

/\* This code is contributed by Devesh Agrawal \*/

length | 1 2 3 4 5 6 7 8

--------------------------------------------

price | 1 5 8 9 10 17 17 20

(¿Máximo valor que puedo obtener por pedazo de longitude n vendiendolo por pedazos?)

// A Dynamic Programming solution for Rod cutting problem

class RodCutting

{

    /\* Returns the best obtainable price for a rod of

       length n and price[] as prices of different pieces \*/

    static int cutRod(int price[],int n)

    {

        int val[] = new int[n+1];

        val[0] = 0;

        // Build the table val[] in bottom up manner and return

        // the last entry from the table

        for (int i = 1; i<=n; i++)

        {

            int max\_val = Integer.MIN\_VALUE;

            for (int j = 0; j < i; j++)

                max\_val = Math.max(max\_val,

                                   price[j] + val[i-j-1]);

            val[i] = max\_val;

        }

        return val[n];

    }

    /\* Driver program to test above functions \*/

    public static void main(String args[])

    {

        int arr[] = new int[] {1, 5, 8, 9, 10, 17, 17, 20};

        int size = arr.length;

        System.out.println("Maximum Obtainable Value is " +

                            cutRod(arr, size));

    }

}

/\* This code is contributed by Rajat Mishra \*/

/\* Dynamic Programming Java implementation of Coin

   Change problem \*/

import java.util.Arrays;

class CoinChange

{

    static long countWays(int S[], int m, int n)

    {

        //Time complexity of this function: O(mn)

        //Space Complexity of this function: O(n)

        // table[i] will be storing the number of solutions

        // for value i. We need n+1 rows as the table is

        // constructed in bottom up manner using the base

        // case (n = 0)

        long[] table = new long[n+1];

        // Initialize all table values as 0

        Arrays.fill(table, 0);   //O(n)

        // Base case (If given value is 0)

        table[0] = 1;

        // Pick all coins one by one and update the table[]

        // values after the index greater than or equal to

        // the value of the picked coin

        for (int i=0; i<m; i++)

            for (int j=S[i]; j<=n; j++)

                table[j] += table[j-S[i]];

        return table[n];

    }

    // Driver Function to test above function

    public static void main(String args[])

    {

        int arr[] = {1, 2, 3};

        int m = arr.length;

        int n = 4;

        System.out.println(countWays(arr, m, n));

    }

}

// This code is contributed by Pankaj Kumar

(Dado un diccionario y un string s, es possible partir s para que pueda ser formado con palabras del diccionario?)

// A Dynamic Programming based program to test whether a given string can

// be segmented into space separated words in dictionary

#include <iostream>

#include <string.h>

using namespace std;

/\* A utility function to check whether a word is present in dictionary or not.

  An array of strings is used for dictionary.  Using array of strings for

  dictionary is definitely not a good idea. We have used for simplicity of

  the program\*/

int dictionaryContains(string word)

{

    string dictionary[] = {"mobile","samsung","sam","sung","man","mango",

                           "icecream","and","go","i","like","ice","cream"};

    int size = sizeof(dictionary)/sizeof(dictionary[0]);

    for (int i = 0; i < size; i++)

        if (dictionary[i].compare(word) == 0)

           return true;

    return false;

}

// Returns true if string can be segmented into space separated

// words, otherwise returns false

bool wordBreak(string str)

{

    int size = str.size();

    if (size == 0)   return true;

    // Create the DP table to store results of subroblems. The value wb[i]

    // will be true if str[0..i-1] can be segmented into dictionary words,

    // otherwise false.

    bool wb[size+1];

    memset(wb, 0, sizeof(wb)); // Initialize all values as false.

    for (int i=1; i<=size; i++)

    {

        // if wb[i] is false, then check if current prefix can make it true.

        // Current prefix is "str.substr(0, i)"

        if (wb[i] == false && dictionaryContains( str.substr(0, i) ))

            wb[i] = true;

        // wb[i] is true, then check for all substrings starting from

        // (i+1)th character and store their results.

        if (wb[i] == true)

        {

            // If we reached the last prefix

            if (i == size)

                return true;

            for (int j = i+1; j <= size; j++)

            {

                // Update wb[j] if it is false and can be updated

                // Note the parameter passed to dictionaryContains() is

                // substring starting from index 'i' and length 'j-i'

                if (wb[j] == false && dictionaryContains( str.substr(i, j-i) ))

                    wb[j] = true;

                // If we reached the last character

                if (j == size && wb[j] == true)

                    return true;

            }

        }

    }

    /\* Uncomment these lines to print DP table "wb[]"

     for (int i = 1; i <= size; i++)

        cout << " " << wb[i]; \*/

    // If we have tried all prefixes and none of them worked

    return false;

}

// Driver program to test above functions

int main()

{

    wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";

    wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";

    wordBreak("")? cout <<"Yes\n": cout << "No\n";

    wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";

    wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";

    wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";

    return 0;

}

(Given a rope of length n meters, cut the rope in different parts of integer lengths in a way that maximizes product of lengths of all parts. You must make at least one cut. Assume that the length of rope is more than 2 meters)

// Java program to find maximum product

import java.io.\*;

class GFG {

    /\* The main function that returns the

    max possible product \*/

    static int maxProd(int n)

    {

    // n equals to 2 or 3 must be handled

    // explicitly

    if (n == 2 || n == 3) return (n-1);

    // Keep removing parts of size 3

    // while n is greater than 4

    int res = 1;

    while (n > 4)

    {

        n -= 3;

        // Keep multiplying 3 to res

        res \*= 3;

    }

    // The last part multiplied by

    // previous parts

    return (n \* res);

    }

    /\* Driver program to test above functions \*/

    public static void main(String[] args)

    {

        System.out.println("Maximum Product is "

                            + maxProd(10));

    }

}

// This code is contributed by Prerna Saini

// A Dynamic Programming solution for Max Product Problem

int maxProd(int n)

{

   int val[n+1];

   val[0] = val[1] = 0;

   // Build the table val[] in bottom up manner and return

   // the last entry from the table

   for (int i = 1; i <= n; i++)

   {

      int max\_val = 0;

      for (int j = 1; j <= i/2; j++)

         max\_val = max(max\_val, (i-j)\*j, j\*val[i-j]);

      val[i] = max\_val;

   }

   return val[n];

}

// C# program to find number

// of ways to get sum 'x'

// with 'n' dice where every

// dice has 'm' faces

using System;

class GFG

{

// The main function that returns

// number of ways to get sum 'x'

// with 'n' dice and 'm' with m faces.

static int findWays(int m,

                    int n, int x)

{

    // Create a table to store

    // results of subproblems.

    // row and column are used

    // for simpilicity (Number

    // of dice is directly used

    // as row index and sum is

    // directly used as column

    // index). The entries in 0th

    // row and 0th column are

    // never used.

    int[,] table = new int[n + 1,

                           x + 1];

    // Initialize all

    // entries as 0

    for (int i = 0; i <= n; i++)

    for (int j = 0; j <= x; j++)

    table[i, j] = 0;

    // Table entries for

    // only one dice

    for (int j = 1;

             j <= m && j <= x; j++)

        table[1, j] = 1;

    // Fill rest of the entries

    // in table using recursive

    // relation i: number of

    // dice, j: sum

    for (int i = 2; i <= n; i++)

        for (int j = 1; j <= x; j++)

            for (int k = 1;

                     k <= m && k < j; k++)

                table[i, j] += table[i - 1,

                                     j - k];

    /\* Uncomment these lines to

    see content of table

    for (int i = 0; i <= n; i++)

    {

    for (int j = 0; j <= x; j++)

        cout << table[i][j] << " ";

    cout << endl;

    } \*/

    return table[n, x];

}

// Driver Code

public static void Main()

{

    Console.WriteLine(findWays(4, 2, 1));

    Console.WriteLine(findWays(2, 2, 3));

    Console.WriteLine(findWays(6, 3, 8));

    Console.WriteLine(findWays(4, 2, 5));

    Console.WriteLine(findWays(4, 3, 5));

}

}

// This code is contributed by mits.

(You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.)

\* Dynamic Programming implementation

of Box Stacking problem in Java\*/

import java.util.\*;

public class GFG {

    /\* Representation of a box \*/

    static class Box implements Comparable<Box>{

        // h --> height, w --> width,

        // d --> depth

        int h, w, d, area;

        // for simplicity of solution,

        // always keep w <= d

        /\*Constructor to initialise object\*/

        public Box(int h, int w, int d) {

            this.h = h;

            this.w = w;

            this.d = d;

        }

        /\*To sort the box array on the basis

        of area in decreasing order of area \*/

        @Override

        public int compareTo(Box o) {

            return o.area-this.area;

        }

    }

    /\* Returns the height of the tallest

    stack that can be formed with give

    type of boxes \*/

    static int maxStackHeight( Box arr[], int n){

        Box[] rot = new Box[n\*3];

        /\* New Array of boxes is created -

        considering all 3 possible rotations,

        with width always greater than equal

        to width \*/

        for(int i = 0;i < n;i++){

            Box box = arr[i];

            /\* Orignal Box\*/

            rot[3\*i] = new Box(box.h, Math.max(box.w,box.d),

                                    Math.min(box.w,box.d));

            /\* First rotation of box\*/

            rot[3\*i + 1] = new Box(box.w, Math.max(box.h,box.d),

                                       Math.min(box.h,box.d));

            /\* Second rotation of box\*/

            rot[3\*i + 2] = new Box(box.d, Math.max(box.w,box.h),

                                       Math.min(box.w,box.h));

        }

        /\* Calculating base area of

        each of the boxes.\*/

        for(int i = 0; i < rot.length; i++)

            rot[i].area = rot[i].w \* rot[i].d;

        /\* Sorting the Boxes on the bases

        of Area in non Increasing order.\*/

        Arrays.sort(rot);

        int count = 3 \* n;

        /\* Initialize msh values for all

        indexes

        msh[i] --> Maximum possible Stack Height

                   with box i on top \*/

        int[]msh = new int[count];

        for (int i = 0; i < count; i++ )

            msh[i] = rot[i].h;

        /\* Computing optimized msh[]

        values in bottom up manner \*/

        for(int i = 0; i < count; i++){

            msh[i] = 0;

            Box box = rot[i];

            int val = 0;

            for(int j = 0; j < i; j++){

                Box prevBox = rot[j];

                if(box.w < prevBox.w && box.d < prevBox.d){

                    val = Math.max(val, msh[j]);

                }

            }

            msh[i] = val + box.h;

        }

        int max = -1;

        /\* Pick maximum of all msh values \*/

        for(int i = 0; i < count; i++){

            max = Math.max(max, msh[i]);

        }

        return max;

    }

    /\* Driver program to test above function \*/

    public static void main(String[] args) {

        Box[] arr = new Box[4];

        arr[0] = new Box(4, 6, 7);

        arr[1] = new Box(1, 2, 3);

        arr[2] = new Box(4, 5, 6);

        arr[3] = new Box(10, 12, 32);

        System.out.println("The maximum possible "+

                           "height of stack is " +

                           maxStackHeight(arr,4));

    }

}

// This code is contributed by Divyam

(The following is a description of the instance of this famous puzzle involving n=2 eggs and a building with k=36 floors.

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

…..An egg that survives a fall can be used again.

…..A broken egg must be discarded.

…..The effect of a fall is the same for all eggs.

…..If an egg breaks when dropped, then it would break if dropped from a higher floor.

…..If an egg survives a fall then it would survive a shorter fall.

…..It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?

The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.)

//A Dynamic Programming based Python Program for the Egg Dropping Puzzle

class EggDrop

{

    // A utility function to get maximum of two integers

    static int max(int a, int b) { return (a > b)? a: b; }

    /\* Function to get minimum number of trials needed in worst

    case with n eggs and k floors \*/

    static int eggDrop(int n, int k)

    {

       /\* A 2D table where entery eggFloor[i][j] will represent minimum

       number of trials needed for i eggs and j floors. \*/

        int eggFloor[][] = new int[n+1][k+1];

        int res;

        int i, j, x;

        // We need one trial for one floor and0 trials for 0 floors

        for (i = 1; i <= n; i++)

        {

            eggFloor[i][1] = 1;

            eggFloor[i][0] = 0;

        }

       // We always need j trials for one egg and j floors.

        for (j = 1; j <= k; j++)

            eggFloor[1][j] = j;

        // Fill rest of the entries in table using optimal substructure

        // property

        for (i = 2; i <= n; i++)

        {

            for (j = 2; j <= k; j++)

            {

                eggFloor[i][j] = Integer.MAX\_VALUE;

                for (x = 1; x <= j; x++)

                {

                     res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]);

                     if (res < eggFloor[i][j])

                        eggFloor[i][j] = res;

                }

            }

        }

        // eggFloor[n][k] holds the result

        return eggFloor[n][k];

    }

    /\* Driver program to test to pront printDups\*/

    public static void  main(String args[] )

    {

        int n = 2, k = 10;

        System.out.println("Minimum number of trials in worst case with "+n+"  eggs and "+k+

                 " floors is "+eggDrop(n, k));

    }

}

/\*This code is contributed by Rajat Mishra\*/