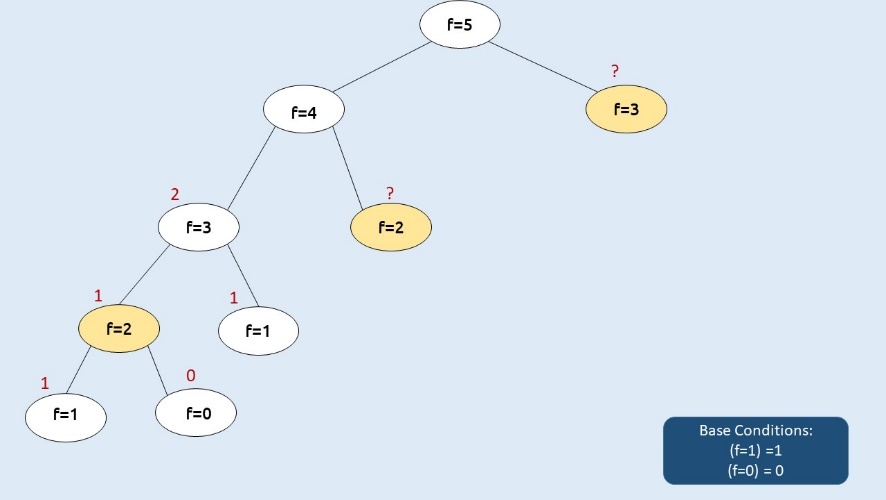
**Dynamic Programming**

**1) Fibonacci Series**

Ex : 0,1,1,2,3,5,8,13,21,...



**Solution :(Memoization )**

import java.util.\*;

class TUF{

static int f(int n, int[] dp){

if(n<=1) return n;

if(dp[n]!= -1) return dp[n];

return dp[n]= f(n-1,dp) + f(n-2,dp);

}

public static void main(String args[]) {

int n=5;

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

Arrays.fill(dp,-1);

System.out.println(f(n,dp));

}

}

**Tabulation :**

import java.util.\*;

class TUF{

public static void main(String args[]) {

int n=5;

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

Arrays.fill(dp,-1);

dp[0]= 0;

dp[1]= 1;

for(int i=2; i<=n; i++){

dp[i] = dp[i-1]+ dp[i-2];

}

System.out.println(dp[n]);

}

}

**Tabulation with 0(1) space Complexity :**

import java.util.\*;

class TUF{

public static void main(String args[]) {

int n=5;

int prev2 = 0;

int prev = 1;

for(int i=2; i<=n; i++){

int cur\_i = prev2+ prev;

prev2 = prev;

prev= cur\_i;

}

System.out.println(prev);

}

}

**2) Climbing Stairs (Leetcode – 70)**

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

Example 1:

Input: n = 2

Output: 2

Explanation: There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps

Example 2:

Input: n = 3

Output: 3

Explanation: There are three ways to climb to the top.

1. 1 step + 1 step + 1 step

2. 1 step + 2 steps

3. 2 steps + 1 step

Constraints:

1 <= n <= 45

**Solution :(fibonaci model - 3 approaches)**

**Recursion :**

class Solution {

public int climbStairs(int n) {

if (n <= 1) return 1; // Case - A (Base Condition)

return (climbStairs(n - 1) + climbStairs(n - 2)); // Case - C (Recurring Condition)

}

}

**Tabulation :**

class Solution {

    public int climbStairs(int n) {

        int[] climbStore = new int[n + 1];

        climbStore[0] = 1; climbStore[1] = 1; // Base Conditions of Recursion

        for (int i = 2; i <= n; i ++) {

            climbStore[i] = climbStore[i - 1] + climbStore[i - 2]; // Recurring Condition of Recursion

        }

        return climbStore[n]; // Final Value

    }

}

**3) Frog Jump (Coding ninjas)**

There is a frog on the '1st' step of an 'N' stairs long staircase. The frog wants to reach the 'Nth' stair. 'HEIGHT[i]' is the height of the '(i+1)th' stair.If Frog jumps from 'ith' to 'jth' stair, the energy lost in the jump is given by absolute value of ( HEIGHT[i-1] - HEIGHT[j-1] ). If the Frog is on 'ith' staircase, he can jump either to '(i+1)th' stair or to '(i+2)th' stair. Your task is to find the minimum total energy used by the frog to reach from '1st' stair to 'Nth' stair.

For Example

If the given ‘HEIGHT’ array is [10,20,30,10], the answer 20 as the frog can jump from 1st stair to 2nd stair (|20-10| = 10 energy lost) and then a jump from 2nd stair to last stair (|10-20| = 10 energy lost). So, the total energy lost is 20.

Detailed explanation ( Input/output format, Notes, Images )

Constraints:

1 <= T <= 10

1 <= N <= 100000.

1 <= HEIGHTS[i] <= 1000 .

Time limit: 1 sec

Sample Input 1:

2

4

10 20 30 10

3

10 50 10

Sample Output 1:

20

0

Explanation of sample input 1:

For the first test case,

The frog can jump from 1st stair to 2nd stair (|20-10| = 10 energy lost).

Then a jump from the 2nd stair to the last stair (|10-20| = 10 energy lost).

So, the total energy lost is 20 which is the minimum.

Hence, the answer is 20.

For the second test case:

The frog can jump from 1st stair to 3rd stair (|10-10| = 0 energy lost).

So, the total energy lost is 0 which is the minimum.

Hence, the answer is 0.

Sample Input 2:

2

8

7 4 4 2 6 6 3 4

6

4 8 3 10 4 4

Sample Output 2:

7

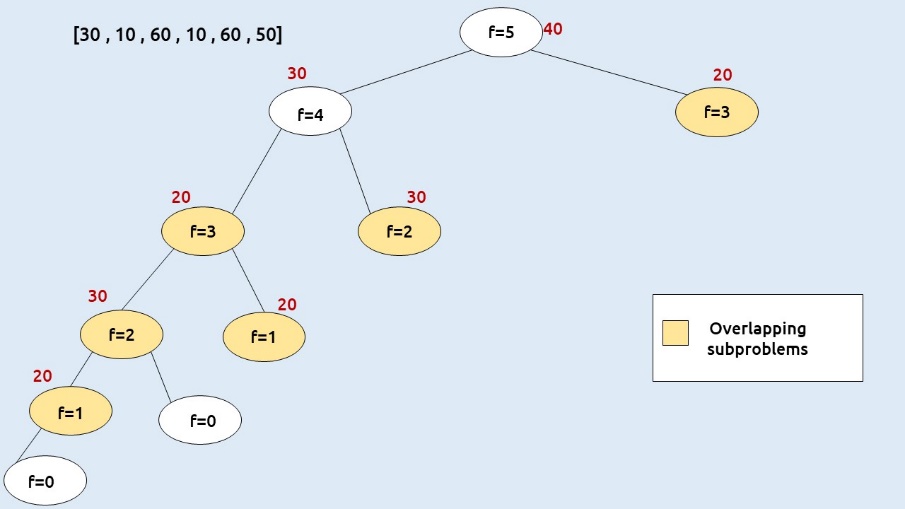
2

Hints:

1. Think about all the possibilities at each stair.

2. Using recursion, try to divide the problem into subproblems and calculate the answer for each subproblem only once - store it for reusing in the future.

3. The above can also be done iteratively.



**Solution :**

**Recursion :**

public static int frogJump(int n, int heights[]) {

// Base case: If there is only one stone, no energy is needed

if (n == 1) return 0;

// Jump from the previous stone

int oneStep = frogJump(n - 1, heights) + Math.abs(heights[n - 1] - heights[n - 2]);

// Jump from two stones behind

int twoSteps = Integer.MAX\_VALUE;

if (n > 2) {

twoSteps = frogJump(n - 2, heights) + Math.abs(heights[n - 1] - heights[n - 3]);

}

// Return the minimum energy required

return Math.min(oneStep, twoSteps);

}

**Memoization :**

static int solve(int ind,int[] height,int[] dp){

if(ind==0) return 0;

if(dp[ind]!=-1) return dp[ind];

int jumpTwo = Integer.MAX\_VALUE;

int jumpOne= solve(ind-1, height,dp)+ Math.abs(height[ind]-height[ind-1]);

if(ind>1)

jumpTwo = solve(ind-2, height,dp)+ Math.abs(height[ind]-height[ind-2]);

return dp[ind]=Math.min(jumpOne, jumpTwo);

}

**Tabulation :**

public static int frogJump(int n, int heights[]) {

int[] dp = new int[n];

Arrays.fill(dp,-1);

dp[0]=0;

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

{

int oneStep = dp[i-1]+Math.abs(heights[i]-heights[i-1]);

int secondStep = Integer.MAX\_VALUE;

if(i>1)

secondStep = dp[i-2] + Math.abs(heights[i]-heights[i-2]);

dp[i]=Math.min(oneStep, secondStep);

}

return dp[n-1];

}

**4) Frog jump with K distance**

The frog was allowed to jump either one or two steps at a time. In this question, the frog is allowed to jump up to ‘K’ steps at a time. If K=4, the frog can jump 1,2,3, or 4 steps at every index.

**Memoization :**

static int solveUtil(int ind, int[] height, int[] dp, int k) {

// Base case: If we are at the beginning (index 0), no cost is needed.

if (ind == 0)

return 0;

// If the result for this index has been previously calculated, return it.

if (dp[ind] != -1)

return dp[ind];

int mmSteps = Integer.MAX\_VALUE;

// Loop to try all possible jumps from '1' to 'k'

for (int j = 1; j <= k; j++) {

// Ensure that we do not jump beyond the beginning of the array

if (ind - j >= 0) {

// Calculate the cost for this jump and update mmSteps with the minimum cost

int jump = solveUtil(ind - j, height, dp, k) + Math.abs(height[ind] - height[ind - j]);

mmSteps = Math.min(jump, mmSteps);

}

}

// Store the minimum cost for this index in the dp array and return it.

return dp[ind] = mmSteps;

}

**Tabulation :**

static int solveUtil(int n, int[] height, int[] dp, int k) {

dp[0] = 0;

// Loop through the array to fill in the dp array

for (int i = 1; i < n; i++) {

int mmSteps = Integer.MAX\_VALUE;

// Loop to try all possible jumps from '1' to 'k'

for (int j = 1; j <= k; j++) {

if (i - j >= 0) {

int jump = dp[i - j] + Math.abs(height[i] - height[i - j]);

mmSteps = Math.min(jump, mmSteps);

}

}

dp[i] = mmSteps;

}

return dp[n - 1]; // The result is stored in the last element of dp

}

**5) Maximum sum of non-adjacent elements (coding ninjas)**

You are given an array/list of ‘N’ integers. You are supposed to return the maximum sum of the subsequence with the constraint that no two elements are adjacent in the given array/list.

Note:

A subsequence of an array/list is obtained by deleting some number of elements (can be zero) from the array/list, leaving the remaining elements in their original order.

Detailed explanation ( Input/output format, Notes, Images )

Constraints:

1 <= T <= 500

1 <= N <= 1000

0 <= ARR[i] <= 10^5

Where 'ARR[i]' denotes the 'i-th' element in the array/list.

Time Limit: 1 sec.

Sample Input 1:

2

3

1 2 4

4

2 1 4 9

Sample Output 1:

5

11

Explanation to Sample Output 1:

In test case 1, the sum of 'ARR[0]' & 'ARR[2]' is 5 which is greater than 'ARR[1]' which is 2 so the answer is 5.

In test case 2, the sum of 'ARR[0]' and 'ARR[2]' is 6, the sum of 'ARR[1]' and 'ARR[3]' is 10, and the sum of 'ARR[0]' and 'ARR[3]' is 11. So if we take the sum of 'ARR[0]' and 'ARR[3]', it will give the maximum sum of sequence in which no elements are adjacent in the given array/list.

Sample Input 2:

2

5

1 2 3 5 4

9

1 2 3 1 3 5 8 1 9

Sample Output 2:

8

24

Explanation to Sample Output 2:

In test case 1, out of all the possibilities, if we take the sum of 'ARR[0]', 'ARR[2]' and 'ARR[4]', i.e. 8, it will give the maximum sum of sequence in which no elements are adjacent in the given array/list.

In test case 2, out of all the possibilities, if we take the sum of 'ARR[0]', 'ARR[2]', 'ARR[4]', 'ARR[6]' and 'ARR[8]', i.e. 24 so, it will give the maximum sum of sequence in which no elements are adjacent in the given array/list.

**Solution :**

**Recursion 1:(another approach not support memorization)**

public static int maxSum(ArrayList<Integer> nums,int start,int sum)

{

int max=sum;

for(int i=start;i<nums.size();i++)

{

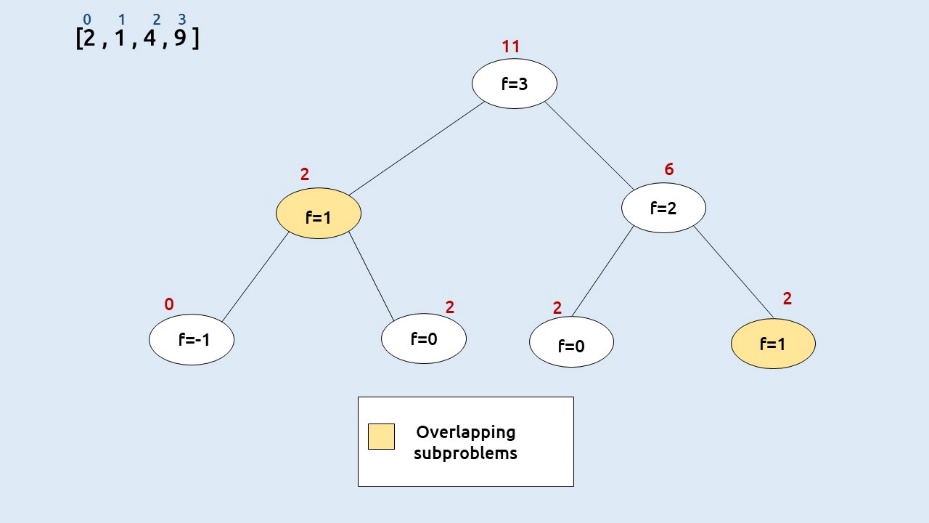
max=Math.max(maxSum(nums,i+2,sum+nums.get(i)),max);

}

return max;

}

**Memoization :**



import java.util.\* ;

import java.io.\*;

import java.util.\*;

public class Solution {

static int solveUtil(int ind, ArrayList<Integer> arr, int[] dp) {

// If the index is negative, there are no elements left to consider.

if (ind < 0) return 0;

// If the index is 0, there is only one element to consider, so return its value.

if (ind == 0) return arr[ind];

// If we have already calculated the result for this index, return it.

if (dp[ind] != -1) return dp[ind];

// Calculate the maximum sum by either picking the current element or not picking it.

int pick = arr.get(ind) + solveUtil(ind - 2, arr, dp);

int nonPick = solveUtil(ind - 1, arr, dp);

// Store the maximum of the two options in the dp array for future reference.

return dp[ind] = Math.max(pick, nonPick);

}

public static int maximumNonAdjacentSum(ArrayList<Integer> nums) {

int dp[] = new int[nums.size()];

// Initialize the dp array with -1 to indicate that values are not calculated yet.

Arrays.fill(dp, -1);

// Call the recursive solver for the last index (n-1).

return solveUtil(nums.size() - 1, nums, dp);

}

}

**Tabulation :**

import java.util.\* ;

import java.io.\*;

import java.util.\*;

public class Solution {

static int solveUtil(int n, ArrayList<Integer> arr, int[] dp) {

// Initialize the dp array with the first element of the input array.

dp[0] = arr.get(0);

// Iterate through the input array to fill the dp array.

for (int i = 1; i < n; i++) {

// Calculate the maximum sum by either picking the current element or not picking it.

int pick = arr.get(i);

// If there are at least two elements before the current element, add the value from dp[i-2].

if (i > 1)

pick += dp[i - 2];

// The non-pick option is to use the maximum sum from the previous element.

int nonPick = dp[i - 1];

// Store the maximum of the two options in the dp array for the current index.

dp[i] = Math.max(pick, nonPick);

}

// The final element of the dp array contains the maximum possible sum.

return dp[n - 1];

}

public static int maximumNonAdjacentSum(ArrayList<Integer> nums) {

int dp[] = new int[nums.size()];

// Initialize the dp array with -1 to indicate that values are not calculated yet.

Arrays.fill(dp, -1);

// Call the recursive solver for the last index (n-1).

return solveUtil(nums.size(), nums, dp);

}

}

**6) House Robber (Leetcode - 198)**

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Example 1:

Input: nums = [1,2,3,1]

Output: 4

Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).

Total amount you can rob = 1 + 3 = 4.

Example 2:

Input: nums = [2,7,9,3,1]

Output: 12

Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).

Total amount you can rob = 2 + 9 + 1 = 12.

Constraints:

1 <= nums.length <= 100

0 <= nums[i] <= 400  
  
**Solution :**

**Memoization :**

class Solution {

    public int robbery(int[] nums,int i,int[] dp)

    {

        if(i<0) return 0;

        if(i==0) return nums[i];

        if(dp[i]!=-1) return dp[i];

        return dp[i]=Math.max(nums[i]+robbery(nums,i-2,dp),robbery(nums,i-1,dp));

    }

    public int rob(int[] nums) {

        int[] dp = new int[nums.length];

        Arrays.fill(dp,-1);

        return robbery(nums,nums.length-1,dp);

    }

}

**Tabulation :**

public int tab(int[] nums,int[] dp)

    {

        dp[0]=nums[0];

        for(int i=1;i<nums.length;i++)

        {

            dp[i]=Math.max(nums[i]+((i>1)?dp[i-2]:0),dp[i-1]);

        }

        return dp[nums.length-1];

    }

//House robber doesn’t contain both first and last a time

In this first call previous function with removing first and then removing last

class TUF{

static long solve(ArrayList<Integer> arr){

    int n = arr.size();

    long prev = arr.get(0);

    long prev2 =0;

    for(int i=1; i<n; i++){

        long pick = arr.get(i);

        if(i>1)

            pick += prev2;

        long nonPick = 0 + prev;

        long cur\_i = Math.max(pick, nonPick);

        prev2 = prev;

        prev= cur\_i;

    }

    return prev;

}

static long robStreet(int n, ArrayList<Integer> arr){

    ArrayList<Integer> arr1=new ArrayList<>();

    ArrayList<Integer> arr2=new ArrayList<>();

    if(n==1)

       return arr.get(0);

    for(int i=0; i<n; i++){

        if(i!=0) arr1.add(arr.get(i));

        if(i!=n-1) arr2.add(arr.get(i));

    }

    long ans1 = solve(arr1);

    long ans2 = solve(arr2);

    return Math.max(ans1,ans2);

}

**7) Ninja’s Training (coding ninjas)**

Ninja is planing this ‘N’ days-long training schedule. Each day, he can perform any one of these three activities. (Running, Fighting Practice or Learning New Moves). Each activity has some merit points on each day. As Ninja has to improve all his skills, he can’t do the same activity in two consecutive days. Can you help Ninja find out the maximum merit points Ninja can earn?

You are given a 2D array of size N\*3 ‘POINTS’ with the points corresponding to each day and activity. Your task is to calculate the maximum number of merit points that Ninja can earn.

For Example

If the given ‘POINTS’ array is [[1,2,5], [3 ,1 ,1] ,[3,3,3] ],the answer will be 11 as 5 + 3 + 3.

Detailed explanation ( Input/output format, Notes, Images )

Constraints:

1 <= T <= 10

1 <= N <= 100000.

1 <= values of POINTS arrays <= 100 .

Time limit: 1 sec

Sample Input 1:

2

3

1 2 5

3 1 1

3 3 3

3

10 40 70

20 50 80

30 60 90

Sample Output 1:

11

210

Explanation of sample input 1:

For the first test case,

One of the answers can be:

On the first day, Ninja will learn new moves and earn 5 merit points.

On the second day, Ninja will do running and earn 3 merit points.

On the third day, Ninja will do fighting and earn 3 merit points.

The total merit point is 11 which is the maximum.

Hence, the answer is 11.

For the second test case:

One of the answers can be:

On the first day, Ninja will learn new moves and earn 70 merit points.

On the second day, Ninja will do fighting and earn 50 merit points.

On the third day, Ninja will learn new moves and earn 90 merit points.

The total merit point is 210 which is the maximum.

Hence, the answer is 210.

Sample Input 2:

2

3

18 11 19

4 13 7

1 8 13

2

10 50 1

5 100 11

Sample Output 2:

45

110

**Solution :**

**Memoization :**

import java.util.\*;

public class Solution {

public static int ninja(int day,int last,int points[][],int dp[][])

{

if(dp[day][last]!=-1) return dp[day][last];

if(day==0)

{

int max=Integer.MIN\_VALUE;

for(int i=0;i<3;i++)

{

if(i!=last)

{

max = Math.max(max,points[0][i]);

}

}

return dp[day][last]=max;

}

int max=Integer.MIN\_VALUE;

for(int i=0;i<3;i++)

{

if(i!=last)

{

int point = points[day][i]+ninja(day-1,i,points,dp);

max = Math.max(point,max);

}

}

return dp[day][last]=max;

}

public static int ninjaTraining(int n, int points[][]) {

int[][] dp = new int[n][4];

for(int[] i:dp)

{

Arrays.fill(i,-1);

}

return ninja(n-1,3,points,dp);

}

}

**Tabulation :**

import java.util.\*;

public class Solution {

public static int tab(int[][] points, int[][] dp) {

dp[0][0] = Math.max(points[0][1], points[0][2]);

dp[0][1] = Math.max(points[0][0], points[0][2]);

dp[0][2] = Math.max(points[0][0], points[0][1]);

dp[0][3] = Math.max(points[0][0], Math.max(points[0][1], points[0][2]));

for (int day = 1; day < dp.length; day++) {

for (int last = 0; last < 4; last++) {

dp[day][last] = 0;

for (int task = 0; task < 3; task++) {

if (last != task) {

int point = points[day][task] + dp[day - 1][task];

dp[day][last] = Math.max(point, dp[day][last]);

}

}

}

}

return dp[dp.length - 1][3];

}

public static int ninjaTraining(int n, int points[][]) {

int[][] dp = new int[n][4];

for(int[] i:dp)

{

Arrays.fill(i,-1);

}

return tab(points,dp);

}

}

**8) Unique Paths (Leetcode - 62)**

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

Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

The test cases are generated so that the answer will be less than or equal to 2 \* 109.

Example 1:



Input: m = 3, n = 7

Output: 28

Example 2:

Input: m = 3, n = 2

Output: 3

Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:

1. Right -> Down -> Down

2. Down -> Down -> Right

3. Down -> Right -> Down

Constraints:

1 <= m, n <= 100

**Solution :**

class Solution {

    //memoization

    public int path(int m,int n,int i,int j,int[][] dp)

    {

        if(i==m-1 && j==n-1) return 1;

        if(i==m || j==n) return 0;

        if(dp[i][j]!=-1) return dp[i][j];

        return dp[i][j]=path(m,n,i+1,j,dp)+path(m,n,i,j+1,dp);

    }

    //tabulation

    public int tab(int m,int n)

    {

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

        dp[m-1][n-1]=1;

        for(int i=m-1;i>=0;i--)

        {

            for(int j=n-1;j>=0;j--)

            {

                if(dp[i][j]==0)

                {

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

                }

            }

        }

        return dp[0][0];

    }

    public int uniquePaths(int m, int n) {

        int dp[][] = new int[m][n];

       for(int[] i:dp)

       {

        Arrays.fill(i,0);

       }

        return tab(m,n);

    }

}

**Bottom – Top :**

import java.util.\*;

class TUF {

// Function to count the number of ways to reach cell (m, n)

static int countWaysUtil(int m, int n, int[][] dp) {

// Loop through each cell in the grid

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

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

// Base condition: If we are at the top-left cell (0, 0), there's one way to reach it.

if (i == 0 && j == 0) {

dp[i][j] = 1;

continue;

}

int up = 0;

int left = 0;

// Calculate the number of ways by moving up (if possible) and left (if possible)

if (i > 0)

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

if (j > 0)

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

// Store the total number of ways to reach the current cell in the DP array

dp[i][j] = up + left;

}

}

// Return the number of ways to reach the bottom-right cell (m-1, n-1)

return dp[m - 1][n - 1];

}

// Function to count the number of ways to reach cell (m, n)

static int countWays(int m, int n) {

// Create a 2D DP array to store the results

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

// Initialize the DP array with -1 to indicate uncomputed values

for (int[] row : dp)

Arrays.fill(row, -1);

// Call the countWaysUtil function to calculate and return the result

return countWaysUtil(m, n, dp);

}

public static void main(String args[]) {

int m = 3;

int n = 2;

// Call the countWays function and print the result

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

}

}

**9) Unique Paths II (Leetcode – 63)**

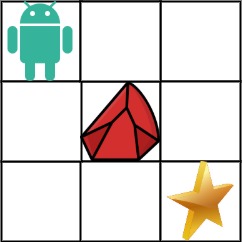
You are given an m x n integer array grid. There is a robot initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

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

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

The testcases are generated so that the answer will be less than or equal to 2 \* 109.

Example 1:



Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]

Output: 2

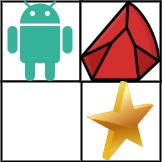
Explanation: There is one obstacle in the middle of the 3x3 grid above.

There are two ways to reach the bottom-right corner:

1. Right -> Right -> Down -> Down

2. Down -> Down -> Right -> Right

Example 2:



Input: obstacleGrid = [[0,1],[0,0]]

Output: 1

Constraints:

m == obstacleGrid.length

n == obstacleGrid[i].length

1 <= m, n <= 100

obstacleGrid[i][j] is 0 or 1.

**Solution :**

class Solution {

    //memoization

    public int path(int[][] grid,int i,int j,int[][] dp)

    {

        if(i==grid.length || j==grid[0].length || grid[i][j]==1) return 0;

        if(i==grid.length-1 && j==grid[0].length-1) return 1;

        if(dp[i][j]!=0) return dp[i][j];

        return dp[i][j]=path(grid,i+1,j,dp)+path(grid,i,j+1,dp);

    }

    //tabulation

    public int tab(int[][] grid)

    {

        int dp[][] =new int[grid.length+1][grid[0].length+1];

        if(grid[grid.length-1][grid[0].length-1]==1)

        {

            return 0;

        }

        dp[grid.length-1][grid[0].length-1]=1;

        for(int i=grid.length-1;i>=0;i--)

        {

            for(int j=grid[0].length-1;j>=0;j--)

            {

                if(grid[i][j]==-1) dp[i][j]=0;

                if(grid[i][j]==0 && dp[i][j]==0)

                {

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

                }

            }

        }

        return dp[0][0];

    }

    public int uniquePathsWithObstacles(int[][] grid) {

        int dp[][] =new int[grid.length+1][grid[0].length+1];

        return path(grid,0,0,dp);

    }

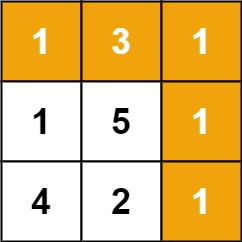
}

**10) Minimum Path Sum (Leetcode - 64)**

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

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

Example 1:



Input: grid = [[1,3,1],[1,5,1],[4,2,1]]

Output: 7

Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

Example 2:

Input: grid = [[1,2,3],[4,5,6]]

Output: 12

Constraints:

m == grid.length

n == grid[i].length

1 <= m, n <= 200

0 <= grid[i][j] <= 200

**Solution :**

class Solution {

    public int min(int[][] grid,int i,int j,int[][] dp)

    {

        if(i==0 && j==0)

        {

            return grid[i][j];

        }

        if(i<0 || j<0)

        {

            return Integer.MAX\_VALUE;

        }

        if(dp[i][j]!=-1) return dp[i][j];

        return dp[i][j]=grid[i][j]+Math.min(min(grid,i-1,j,dp),min(grid,i,j-1,dp));

    }

    public int tab(int[][] grid,int n,int m)

    {

        for(int i=0;i<=n;i++)

        {

            for(int j=0;j<=m;j++)

            {

                if(i==0 && j==0 ) continue;

                int up = Integer.MAX\_VALUE;

                int left = Integer.MAX\_VALUE;

                if(i-1>=0)

                {

                    up = grid[i-1][j];

                }

                if(j-1>=00)

                {

                    left = grid[i][j-1];

                }

                grid[i][j]=grid[i][j]+Math.min(up,left);

            }

        }

         return grid[n][m];

    }

    public int minPathSum(int[][] grid) {

        int n=grid.length;

        int m=grid[0].length;

        // int dp[][] = new int[n][m];

        // for(int[] i:dp)

        // {

        //      Arrays.fill(i,-1);

        // }

        return tab(grid,n-1,m-1);

    }

}

**11) Triangle (Leetcode – 120)**

Given a triangle array, return the minimum path sum from top to bottom.

For each step, you may move to an adjacent number of the row below. More formally, if you are on index i on the current row, you may move to either index i or index i + 1 on the next row.

Example 1:

Input: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]

Output: 11

Explanation: The triangle looks like:

2

3 4

6 5 7

4 1 8 3

The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11 (underlined above).

Example 2:

Input: triangle = [[-10]]

Output: -10

Constraints:

1 <= triangle.length <= 200

triangle[0].length == 1

triangle[i].length == triangle[i - 1].length + 1

-104 <= triangle[i][j] <= 104

**Solution :**

class Solution {

    public int min(List<List<Integer>> triangle,int i,int j,int[][] dp)

    {

        if(i==triangle.size()-1) return triangle.get(i).get(j);

        if(dp[i][j]!=-1) return dp[i][j];

        return dp[i][j]=triangle.get(i).get(j)+Math.min(min(triangle,i+1,j,dp),min(triangle,i+1,j+1,dp));

    }

    public int tab(List<List<Integer>> triangle,int n)

    {

        int dp[][] = new int[n][n];

        for (int j = 0; j < n; j++) {

            dp[n - 1][j] = triangle.get(n-1).get(j);

        }

       for (int i = n - 2; i >= 0; i--) {

            for (int j = i; j >= 0; j--) {

                // Calculate the two possible paths: moving down or moving diagonally

                int down = triangle.get(i).get(j) + dp[i + 1][j];

                int diagonal = triangle.get(i).get(j) + dp[i + 1][j + 1];

                // Store the minimum of the two paths in dp

                dp[i][j] = Math.min(down, diagonal);

            }

        }

        return dp[0][0];

    }

    public int minimumTotal(List<List<Integer>> triangle) {

        int n = triangle.size();

        // int dp[][] = new int[n][n];

        // for(int[] i:dp)

        // {

        //       Arrays.fill(i,-1);

        // }

        return tab(triangle,n);

    }

}

**12) Maximum Path Sum in the matrix (coding ninjas)**

You have been given an N\*M matrix filled with integer numbers, find the maximum sum that can be obtained from a path starting from any cell in the first row to any cell in the last row.

From a cell in a row, you can move to another cell directly below that row, or diagonally below left or right. So from a particular cell (row, col), we can move in three directions i.e.

Down: (row+1,col)

Down left diagonal: (row+1,col-1)

Down right diagonal: (row+1, col+1)

Detailed explanation ( Input/output format, Notes, Images )

Constraints :

1 <= T <= 50

1 <= N <= 100

1 <= M <= 100

-10^4 <= matrix[i][j] <= 10^4

Where 'T' is the number of test cases.

Where 'N' is the number of rows in the given matrix, and 'M' is the number of columns in the given matrix.

And, matrix[i][j] denotes the value at (i,j) cell in the matrix.

Time Limit: 1sec

Input 1 :

2

4 4

1 2 10 4

100 3 2 1

1 1 20 2

1 2 2 1

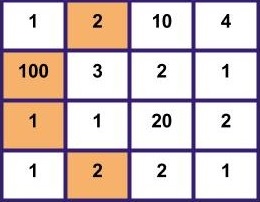
3 3

10 2 3

3 7 2

8 1 5

Output 1 :



105

25

Explanation Of Input 1 :

In the first test case for the given matrix,

The maximum path sum will be 2->100->1->2, So the sum is 105(2+100+1+2).

In the second test case for the given matrix, the maximum path sum will be 10->7->8, So the sum is 25(10+7+8).

Input 2 :

2

3 3

1 2 3

9 8 7

4 5 6

4 6

10 10 2 -13 20 4

1 -9 -81 30 2 5

0 10 4 -79 2 -10

1 -5 2 20 -11 4

Output 2 :

17

74

Explanation Of Input 2 :

In the first test case for the given matrix, the maximum path sum will be 3->8->6, So the sum is 17(3+8+6).

In the second test case for the given matrix, the maximum path sum will be 20->30->4->20, So the sum is 74(20+30+4+20).

**Solution** :

import java.util.\* ;

import java.io.\*;

public class Solution {

    public static int memo(int[][] matrix,int i,int j,int[][] dp)

    {

        if(j<0 || j>=matrix[0].length)

        {

            return Integer.MIN\_VALUE;

        }

        if(i==0) return matrix[0][j];

        if(dp[i][j]!=-1) return dp[i][j];

        int f = memo(matrix,i-1,j-1,dp);

        int s = memo(matrix,i-1,j,dp);

        int t = memo(matrix,i-1,j+1,dp);

        return  dp[i][j]=matrix[i][j]+Math.max(f,Math.max(s,t));

    }

    static int tab(int[][] matrix) {

        int n = matrix.length;

        int m = matrix[0].length;

        int dp[][] = new int[n][m];

        for (int j = 0; j < m; j++) {

            dp[0][j] = matrix[0][j];

        }

        for (int i = 1; i < n; i++) {

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

                int up = matrix[i][j] + dp[i - 1][j];

                int leftDiagonal = matrix[i][j];

                if (j - 1 >= 0) {

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

                } else {

                    leftDiagonal += (int) Math.pow(-10, 9);

                }

                int rightDiagonal = matrix[i][j];

                if (j + 1 < m) {

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

                } else {

                    rightDiagonal += (int) Math.pow(-10, 9);

                }

                // Store the maximum of the three paths in dp

                dp[i][j] = Math.max(up, Math.max(leftDiagonal, rightDiagonal));

            }

        }

        // Find the maximum value in the last row of dp

        int maxi = Integer.MIN\_VALUE;

        for (int j = 0; j < m; j++) {

            maxi = Math.max(maxi, dp[n - 1][j]);

        }

        return maxi;

    }

    public static int getMaxPathSum(int[][] matrix) {

        int max=Integer.MIN\_VALUE;

        int[][] dp = new int[matrix.length][matrix[0].length];

        for(int[] i:dp)

        {

            Arrays.fill(i,-1);

        }

        for(int i=0;i<matrix[0].length;i++)

        {

            max=Math.max(max,memo(matrix,matrix.length-1,i,dp));

        }

        return max;

    }

}

**13) Cherry Pickup II (Leetcode -1463)**

You are given a rows x cols matrix grid representing a field of cherries where grid[i][j] represents the number of cherries that you can collect from the (i, j) cell.

You have two robots that can collect cherries for you:

Robot #1 is located at the top-left corner (0, 0), and

Robot #2 is located at the top-right corner (0, cols - 1).

Return the maximum number of cherries collection using both robots by following the rules below:

From a cell (i, j), robots can move to cell (i + 1, j - 1), (i + 1, j), or (i + 1, j + 1).

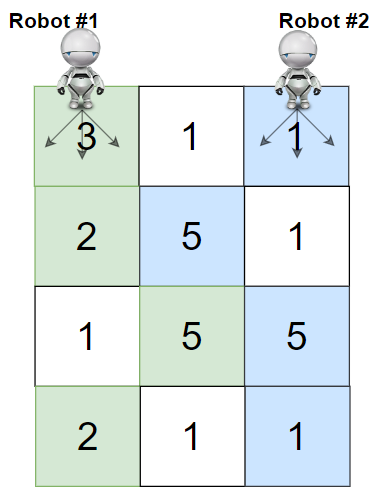
When any robot passes through a cell, It picks up all cherries, and the cell becomes an empty cell.

When both robots stay in the same cell, only one takes the cherries.

Both robots cannot move outside of the grid at any moment.

Both robots should reach the bottom row in grid.

Example 1:



Input: grid = [[3,1,1],[2,5,1],[1,5,5],[2,1,1]]

Output: 24

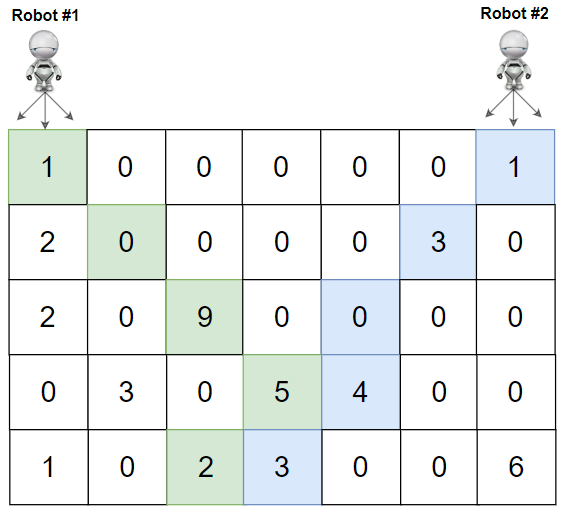
Explanation: Path of robot #1 and #2 are described in color green and blue respectively.

Cherries taken by Robot #1, (3 + 2 + 5 + 2) = 12.

Cherries taken by Robot #2, (1 + 5 + 5 + 1) = 12.

Total of cherries: 12 + 12 = 24.

Example 2:



Input: grid = [[1,0,0,0,0,0,1],[2,0,0,0,0,3,0],[2,0,9,0,0,0,0],[0,3,0,5,4,0,0],[1,0,2,3,0,0,6]]

Output: 28

Explanation: Path of robot #1 and #2 are described in color green and blue respectively.

Cherries taken by Robot #1, (1 + 9 + 5 + 2) = 17.

Cherries taken by Robot #2, (1 + 3 + 4 + 3) = 11.

Total of cherries: 17 + 11 = 28.

Constraints:

rows == grid.length

cols == grid[i].length

2 <= rows, cols <= 70

0 <= grid[i][j] <= 100

**Solution :**

class Solution {

    public int pick(int[][] grid,int i,int j1,int j2,int n,int m,int[][][] dp)

    {

        if(j1<0 || j1>=m || j2<0 || j2>=m ) return Integer.MIN\_VALUE;

        if(i==n-1)

        {

            if(j1==j2) return grid[i][j1];

            else

            {

                return grid[i][j1]+grid[i][j2];

            }

        }

        if(dp[i][j1][j2]!=-1)return dp[i][j1][j2];

        int maxi=Integer.MIN\_VALUE;

        for(int di=-1;di<=1;di++)

        {

            for(int dj=-1;dj<=1;dj++)

            {

                int ans=0;

                if(j1==j2)

                {

                    ans=grid[i][j1]+pick(grid,i+1,j1+di,j2+dj,n,m,dp);

                }

                else

                {

                     ans=grid[i][j1]+grid[i][j2]+pick(grid,i+1,j1+di,j2+dj,n,m,dp);

                }

                maxi=Math.max(ans,maxi);

            }

        }

        return dp[i][j1][j2]=maxi;

    }

    public int tab(int n,int m,int[][] grid,int[][][] dp)

    {

        for(int j1=0;j1<m;j1++)

        {

            for(int j2=0;j2<m;j2++)

            {

                if(j1==j2)

                {

                    dp[n-1][j1][j2]=grid[n-1][j1];

                }

                else

                {

                    dp[n-1][j1][j2]=grid[n-1][j1]+grid[n-1][j2];

                }

            }

        }

        for(int i=n-2;i>=0;i--)

        {

            for(int j1=0;j1<m;j1++ )

            {

                for(int j2=0;j2<m;j2++)

                {

                    int maxi = Integer.MIN\_VALUE;

                     // Inner nested loops to try out 9 options

                     for (int di = -1; di <= 1; di++) {

                        for (int dj = -1; dj <= 1; dj++) {

                            int ans;

                            if (j1 == j2)

                                ans = grid[i][j1];

                            else

                                ans = grid[i][j1] + grid[i][j2];

                            // Check if the indices are valid

                            if ((j1 + di < 0 || j1 + di >= m) || (j2 + dj < 0 || j2 + dj >= m))

                                ans += (int) Math.pow(-10, 9);

                            else

                                ans += dp[i + 1][j1 + di][j2 + dj];

                            // Update maxi with the maximum result

                            maxi = Math.max(ans, maxi);

                        }

                    }

                    dp[i][j1][j2] = maxi;

                }

            }

        }

        return dp[0][0][m - 1];

    }

    public int cherryPickup(int[][] grid) {

        int n=grid.length;

        int m=grid[0].length;

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

        for (int row1[][] : dp) {

            for (int row2[] : row1) {

                Arrays.fill(row2, -1);

            }

        }

        return tab(n,m,grid,dp);

    }

}

**14) Subset sum equal to target**

**Solution :**

static boolean subsetSumUtil(int ind, int target, int[] arr, int[][] dp) {

*// If the target sum is achieved, return true*

if (target == 0)

return true;

*// If we have considered all elements but haven't reached the target, return false*

if (ind == 0)

return arr[0] == target;

*// If the result for this subproblem has already been calculated, return it*

if (dp[ind][target] != -1)

return dp[ind][target] == 0 ? false : true;

*// Try not taking the current element*

boolean notTaken = subsetSumUtil(ind - 1, target, arr, dp);

*// Try taking the current element if it doesn't exceed the target*

boolean taken = false;

if (arr[ind] <= target)

taken = subsetSumUtil(ind - 1, target - arr[ind], arr, dp);

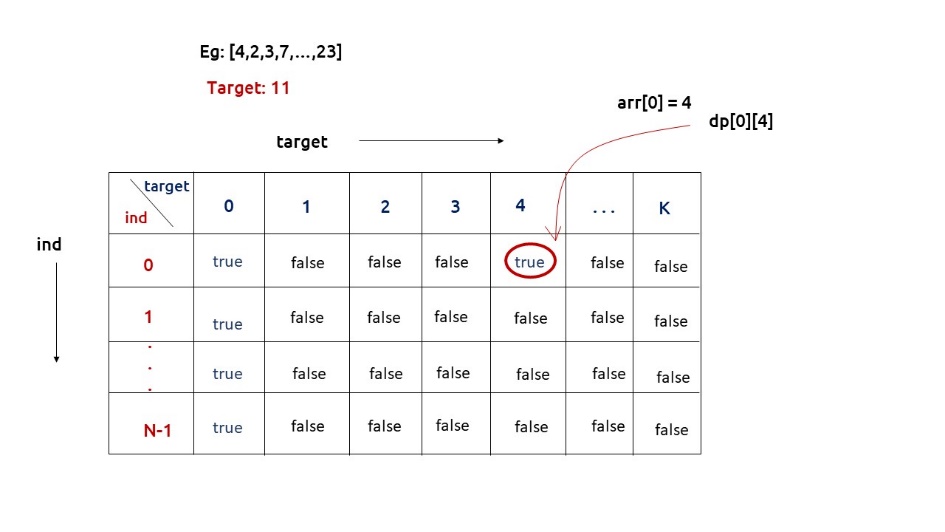
*// Store the result in the DP table and return whether either option was successful*

dp[ind][target] = notTaken || taken ? 1 : 0;

return notTaken || taken;

}

**Tabulation :**



static boolean subsetSumToK(int n, int k, int[] arr) {

*// Create a boolean DP table with dimensions [n][k+1]*

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

*// Initialize the first row of the DP table*

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

dp[i][0] = true;

}

*// Initialize the first column of the DP table*

if (arr[0] <= k) {

dp[0][arr[0]] = true;

}

*// Fill in the DP table using bottom-up approach*

for (int ind = 1; ind < n; ind++) {

for (int target = 1; target <= k; target++) {

*// Calculate if the current target can be achieved without taking the current element*

boolean notTaken = dp[ind - 1][target];

*// Calculate if the current target can be achieved by taking the current element*

boolean taken = false;

if (arr[ind] <= target) {

taken = dp[ind - 1][target - arr[ind]];

}

*// Store the result in the DP table*

dp[ind][target] = notTaken || taken;

}

}

*// The final result is stored in the bottom-right cell of the DP table*

return dp[n - 1][k];

}

**15) Partition Equal Subset Sum (Leetcode - 416)**

Given an integer array nums, return true if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or false otherwise.

Example 1:

Input: nums = [1,5,11,5]

Output: true

Explanation: The array can be partitioned as [1, 5, 5] and [11].

Example 2:

Input: nums = [1,2,3,5]

Output: false

Explanation: The array cannot be partitioned into equal sum subsets.

Constraints:

1 <= nums.length <= 200

1 <= nums[i] <= 100

**Solution :**

class Solution {

    public boolean subSum(int idx,int target,int[] nums,int[][] dp)

    {

        if(target==0)

        {

            return true;

        }

        if(idx==0)

        {

            return nums[0]==target;

        }

        if(dp[idx][target]!=-1) return (dp[idx][target]==0)?false:true;

        boolean pick = false;

        if(target>=nums[idx])

        {

            pick=subSum(idx-1,target-nums[idx],nums,dp);

        }

        boolean notPick = subSum(idx-1,target,nums,dp);

        dp[idx][target]=(pick || notPick)?1:0;

        return  pick||notPick;

    }

    public boolean tab(int[] nums,int target)

    {

        boolean[][] dp = new boolean[nums.length][target+1];

        for(int i=0;i<nums.length;i++) dp[i][0]=true;

        if(nums[0]<=target) dp[0][nums[0]]=true;

        for(int i=1;i<nums.length;i++)

        {

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

            {

                boolean pick=false;

                if(nums[i]<=j)

                {

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

                }

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

                dp[i][j]=pick||notPick;

            }

        }

        return dp[nums.length-1][target];

    }

    public boolean canPartition(int[] nums) {

        int sum=0;

        for(int i:nums)

        {

            sum+=i;

        }

        if((sum&1)==1)//odd return false

        {

            return false;

        }

        int target=sum/2;

//find target sub set is in array

        // int[][] dp = new int[nums.length][target+1];

        // for(int[] i:dp)

        // {

        //     Arrays.fill(i,-1);

        // }

        // if(subSum(nums.length-1,target,nums,dp))

        // {

        //     return true;

        // }

        if(tab(nums,target))

        {

            return true;

        }

        return false;

    }

}

**16) Longest Increasing Subsequence (Leetcode - 300)**

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

Example 1:

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

Output: 4

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

Example 2:

Input: nums = [0,1,0,3,2,3]

Output: 4

Example 3:

Input: nums = [7,7,7,7,7,7,7]

Output: 1

Constraints:

1 <= nums.length <= 2500

-104 <= nums[i] <= 104

Follow up: Can you come up with an algorithm that runs in O(n log(n)) time complexity?

**Solution :**

**Memoization :**

static int getAns(int arr[], int n, int ind, int prev\_index, int[][] dp) {

// Base condition

if (ind == n) {

return 0;

}

if (dp[ind][prev\_index + 1] != -1) {

return dp[ind][prev\_index + 1];

}

int notTake = 0 + getAns(arr, n, ind + 1, prev\_index, dp);

int take = 0;

if (prev\_index == -1 || arr[ind] > arr[prev\_index]) {

take = 1 + getAns(arr, n, ind + 1, ind, dp);

}

dp[ind][prev\_index + 1] = Math.max(notTake, take);

return dp[ind][prev\_index + 1];

}

**DP O(N\*N)**

class Solution {

    static int lis(int arr[], int n)

    {

        int lis[] = new int[n];

        int i, j, max =1;

       lis[0]=1;

        for (i = 1; i < n; i++)

        {

            for (j = 0;j<i; j++)

            {

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

                {

                    lis[i] = lis[j] ;

                }

            }

            lis[i]+=1;

            max=Math.max(max,lis[i]);

        }

        return max;

    }

    public int lengthOfLIS(int[] nums) {

        return lis(nums,nums.length);

    }

}

**Printing longest increasing subsequence : (using tab)**

**//using previous array**

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

    {

        int lis[] = new int[n];

        int prev[] = new int[n];

        int i, j, max = 0,maxInd=0;

       lis[0]=1;

        for (i = 1; i < n; i++)

        {

            prev[i]=i;//initialize previous with same index

            for (j = 0;j<i; j++)

            {

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

                {

                    lis[i] = lis[j] ;

                    prev[i]=j;//updating prev

                }

            }

            lis[i]+=1;

            if(max<lis[i])

            {

                max=lis[i];

                maxInd=i;

            }

        }

          ArrayList<Integer> list = new ArrayList<>();

          list.add(arr[maxInd]);

          while(prev[maxInd]!=maxInd)

          {

            maxInd=prev[maxInd];

            list.add(arr[maxInd]);

          }

        System.out.println(list);

        return max;

    }

    public int lengthOfLIS(int[] nums) {

        return lis(nums,nums.length);

    }

**Binary Search :(O(N\*logN)**

class Solution {

    public int lengthOfLIS(int[] nums) {

        List<Integer> lis = new ArrayList<>(nums.length);

        for (int n : nums) {

            int i = Collections.binarySearch(lis, n);

            //if element not there it returns -ve index of where ele should be present

            if (i < 0) i = -i - 1;

            if (i == lis.size())

                lis.add(n);

            else

                lis.set(i, n);

        }

System.out.println(lis);// **printing the Longest increasing subsequence**

        return lis.size();

    }

}

**17) Maximum Subarray (Leetcode - 53)**

Given an integer array nums, find the

subarray

with the largest sum, and return its sum.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: The subarray [4,-1,2,1] has the largest sum 6.

Example 2:

Input: nums = [1]

Output: 1

Explanation: The subarray [1] has the largest sum 1.

Example 3:

Input: nums = [5,4,-1,7,8]

Output: 23

Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.

Constraints:

1 <= nums.length <= 105

-104 <= nums[i] <= 104

Follow up: If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

**Solution : (Kadane’s Algo)**

class Solution {

    public int maxSubArray(int[] nums) {

        int sum=0,max=Integer.MIN\_VALUE;

        for(int i:nums)

        {

            sum+=i;

            max=Math.max(max,sum);

            if(sum<0)

            {

                sum=0;

            }

        }

        return max;

    }

}

**18) Array partition with minimum difference (coding ninjas)**

You are given an array 'arr' containing 'n' non-negative integers.

Your task is to partition this array into two subsets such that the absolute difference between subset sums is minimum.

You just need to find the minimum absolute difference considering any valid division of the array elements.

Note:

1. Each array element should belong to exactly one of the subsets.

2. Subsets need not always be contiguous.

For example, for the array : [1, 2, 3], some of the possible divisions are

a) {1,2} and {3}

b) {1,3} and {2}.

3. Subset-sum is the sum of all the elements in that subset.

Example:

Input: 'n' = 5, 'arr' = [3, 1, 5, 2, 8].

Ouput: 1

Explanation: We can partition the given array into {3, 1, 5} and {2, 8}.

This will give us the minimum possible absolute difference i.e. (10 - 9 = 1).

Detailed explanation ( Input/output format, Notes, Images )

Sample Input 1:

4

1 2 3 4

Sample Output 1:

0

Explanation for sample input 1:

We can partition the given array into {2,3} and {1,4}.

This will give us the minimum possible absolute difference i.e. (5 - 5 = 0) in this case.

Sample Input 2:

3

8 6 5

Sample Output 2:

3

Explanation for sample input 2:

We can partition the given array into {8} and {6,5}.

This will give us the minimum possible absolute difference i.e. (11 - 8 = 3).

Expected time complexity:

The expected time complexity is O(n \* 𝚺 'arr'[i]), where 𝚺 'arr'[i] denotes the sum of all elements in 'arr'.

Constraints:

1 <= 'n' <= 10^3

0 <= 'arr'[i] <= 10^3

0 <= 𝚺 'arr'[i] <= 10^4,

where 𝚺 'arr'[i] denotes the sum of all elements in 'arr'.

Time Limit: 1sec

**Solution :**

import java.util.\*;

public class Solution {

    public static void subSum(int[] arr,int n,int sum,boolean[][] dp)

    {

        for(int i=0;i<n;i++) dp[i][0]=true;

        if(arr[0]<=sum) dp[0][arr[0]]=true;

        for(int i=1;i<n;i++)

        {

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

            {

                boolean pick = dp[i-1][target];

                boolean notPick=false;

                if(arr[i]<=target)

                {

                    notPick=dp[i-1][target-arr[i]];

                }

                dp[i][target]=pick||notPick;

            }

        }

    }

    public static int minSubsetSumDifference(int []arr, int n) {

        int sum=0;

        for(int i:arr)

        {

            sum+=i;

        }

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

        //subset sum equal to k

        //in dp last row contains 0-targetsum is possible or not

        subSum(arr,n,sum,dp);

        int min=Integer.MAX\_VALUE;

        for(int target=0;target<=sum/2;target++)

        {

            if(dp[n-1][target])

            {

                int s1=target;

                int s2=sum-s1;

                min=Math.min(min,Math.abs(s1-s2));

            }

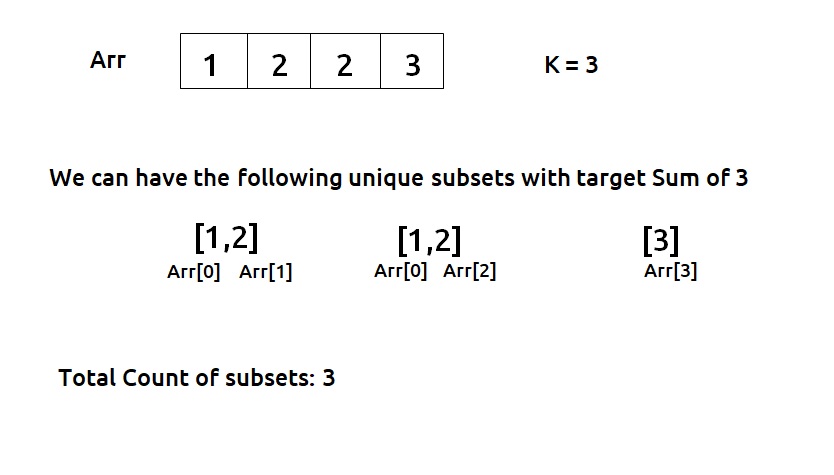
        }

        return min;

    }

}

**19) Count Subsets With Sum K**



**Solution :**

**Memoization :**

static int findWaysUtil(int ind, int target, int[] arr, int[][] dp) {

if (target == 0)

return 1;

if (ind == 0)

return arr[0] == target ? 1 : 0;

if (dp[ind][target] != -1)

return dp[ind][target];

*// Calculate the number of ways when the current element is not taken*

int notTaken = findWaysUtil(ind - 1, target, arr, dp);

*// Calculate the number of ways when the current element is taken*

int taken = 0;

if (arr[ind] <= target)

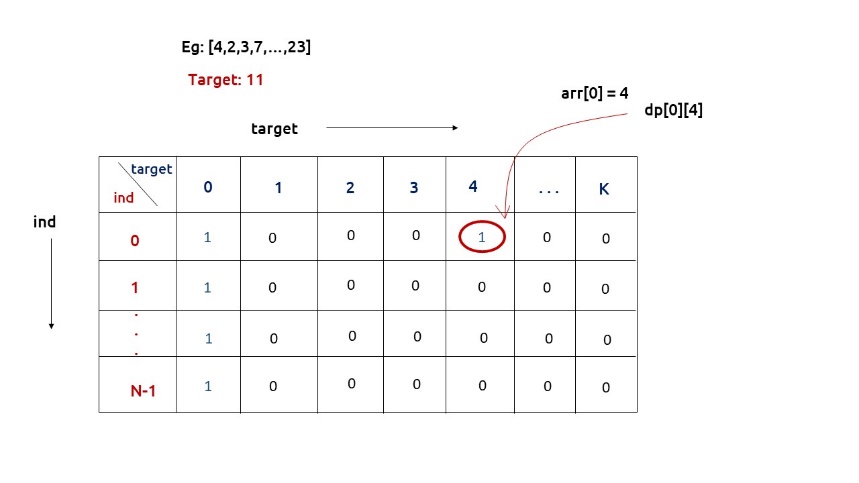
taken = findWaysUtil(ind - 1, target - arr[ind], arr, dp);

*// Store and return the result for the current state*

return dp[ind][target] = notTaken + taken;

}

**Tabulation :**



static int findWays(int[] num, int k) {

int n = num.length;

*// Create a 2D DP array to store the number of ways to achieve each target sum*

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

*// Initialize the first row of the DP array*

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

dp[i][0] = 1;

}

*// Initialize the first column of the DP array*

if (num[0] <= k) {

dp[0][num[0]] = 1;

}

*// Fill in the DP array using bottom-up dynamic programming*

for (int ind = 1; ind < n; ind++) {

for (int target = 1; target <= k; target++) {

*// Calculate the number of ways when the current element is not taken*

int notTaken = dp[ind - 1][target];

*// Calculate the number of ways when the current element is taken*

int taken = 0;

if (num[ind] <= target) {

taken = dp[ind - 1][target - num[ind]];

}

*// Update the DP array for the current element and target sum*

dp[ind][target] = notTaken + taken;

}

}

*// The result is stored in the last cell of the DP array*

return dp[n - 1][k];

}

**20)Partitions With Given Difference (coding ninjas)**

Given an array ‘ARR’, partition it into two subsets (possibly empty) such that their union is the original array. Let the sum of the elements of these two subsets be ‘S1’ and ‘S2’.

Given a difference ‘D’, count the number of partitions in which ‘S1’ is greater than or equal to ‘S2’ and the difference between ‘S1’ and ‘S2’ is equal to ‘D’. Since the answer may be too large, return it modulo ‘10^9 + 7’.

If ‘Pi\_Sj’ denotes the Subset ‘j’ for Partition ‘i’. Then, two partitions P1 and P2 are considered different if:

1) P1\_S1 != P2\_S1 i.e, at least one of the elements of P1\_S1 is different from P2\_S2.

2) P1\_S1 == P2\_S2, but the indices set represented by P1\_S1 is not equal to the indices set of P2\_S2. Here, the indices set of P1\_S1 is formed by taking the indices of the elements from which the subset is formed.

Refer to the example below for clarification.

Note that the sum of the elements of an empty subset is 0.

For example :

If N = 4, D = 3, ARR = {5, 2, 5, 1}

There are only two possible partitions of this array.

Partition 1: {5, 2, 1}, {5}. The subset difference between subset sum is: (5 + 2 + 1) - (5) = 3

Partition 2: {5, 2, 1}, {5}. The subset difference between subset sum is: (5 + 2 + 1) - (5) = 3

These two partitions are different because, in the 1st partition, S1 contains 5 from index 0, and in the 2nd partition, S1 contains 5 from index 2.

Input Format :

The first line contains a single integer ‘T’ denoting the number of test cases, then each test case follows:

The first line of each test case contains two space-separated integers, ‘N’ and ‘D,’ denoting the number of elements in the array and the desired difference.

The following line contains N integers denoting the space-separated integers ‘ARR’.

Output Format :

For each test case, find the number of partitions satisfying the above conditions modulo 10^9 + 7.

Output for each test case will be printed on a separate line.

Note :

You are not required to print anything; it has already been taken care of. Just implement the function.

Constraints :

1 ≤ T ≤ 10

1 ≤ N ≤ 50

0 ≤ D ≤ 2500

0 ≤ ARR[i] ≤ 50

Time limit: 1 sec

Sample Input 1 :

2

4 3

5 2 6 4

4 0

1 1 1 1

Sample Output 1 :

1

6

Explanation For Sample Input 1 :

For test case 1:

We will print 1 because :

There is only one possible partition of this array.

Partition : {6, 4}, {5, 2}. The subset difference between subset sum is: (6 + 4) - (5 + 2) = 3

For test case 2:

We will print 6 because :

The partition {1, 1}, {1, 1} is repeated 6 times:

Partition 1 : {ARR[0], ARR[1]}, {ARR[2], ARR[3]}

Partition 2 : {ARR[0], ARR[2]}, {ARR[1], ARR[3]}

Partition 3 : {ARR[0], ARR[3]}, {ARR[1], ARR[2]}

Partition 4 : {ARR[1], ARR[2]}, {ARR[0], ARR[3]}

Partition 5 : {ARR[1], ARR[3]}, {ARR[0], ARR[2]}

Partition 6 : {ARR[2], ARR[3]}, {ARR[0], ARR[1]}

The difference is in the indices chosen for the subset S1(or S2).

Sample Input 2 :

3

3 1

4 6 3

5 0

3 1 1 2 1

5 1

3 2 2 5 1

Sample Output 2 :

1

6

3

**Solution :**

**//Memoization**

static int mod =(int)(Math.pow(10,9)+7);

static int countPartitionsUtil(int ind, int target,int[] arr, int[][] dp){

if(ind == 0){

if(target==0 && arr[0]==0)

return 2;

if(target==0 || target == arr[0])

return 1;

return 0;

}

if(dp[ind][target]!=-1)

return dp[ind][target];

int notTaken = countPartitionsUtil(ind-1,target,arr,dp);

int taken = 0;

if(arr[ind]<=target)

taken = countPartitionsUtil(ind-1,target-arr[ind],arr,dp);

return dp[ind][target]= (notTaken + taken)%mod;

}

static int countPartitions(int d,int[] arr){

int n = arr.length;

int totSum = 0;

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

totSum += arr[i];

}

*//Checking for edge cases*

if(totSum-d<0) return 0;

if((totSum-d)%2==1) return 0;

int s2 = (totSum-d)/2;

int dp[][] = new int[n][s2+1];

for(int row[]: dp)

Arrays.fill(row,-1);

return countPartitionsUtil(n-1,s2,arr,dp);

}

**Tabulation :**

static int mod =(int)(Math.pow(10,9)+7);

static int findWays(int[] num, int tar){

int n = num.length;

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

if(num[0] == 0) dp[0][0] =2; *// 2 cases -pick and not pick*

else dp[0][0] = 1; *// 1 case - not pick*

if(num[0]!=0 && num[0]<=tar) dp[0][num[0]] = 1; *// 1 case -pick*

for(int ind = 1; ind<n; ind++){

for(int target= 0; target<=tar; target++){

int notTaken = dp[ind-1][target];

int taken = 0;

if(num[ind]<=target)

taken = dp[ind-1][target-num[ind]];

dp[ind][target]= (notTaken + taken)%mod;

}

}

return dp[n-1][tar];

}

static int countPartitions(int n, int d,int[] arr){

int totSum = 0;

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

totSum += arr[i];

}

*//Checking for edge cases*

if(totSum-d <0 || (totSum-d)%2==1 ) return 0;

return findWays(arr,(totSum-d)/2);

}

**21) 0 1 Knapsack (codingNinja)(another model Fractional knapsack)**

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are N items and the ith item weighs wi and is of value vi. Considering the constraints of the maximum weight that a knapsack can carry, you have to find and return the maximum value that a thief can generate by stealing items.

Detailed explanation ( Input/output format, Notes, Images )

Constraints:

1 <= T <= 10

1 <= N <= 10^2

1<= wi <= 50

1 <= vi <= 10^2

1 <= W <= 10^3

Time Limit: 1 second

Sample Input:

1

4 //no of elements

1 2 4 5 //weight

5 4 8 6 //value

5

Sample Output:

13

**Solution :**

import java.util.\* ;

import java.io.\*;

public class Solution{

    static int memo(int idx,int maxWeight,int[] weight,int[] value,int[][] dp)

    {

        if(idx==0)

        {

            if(weight[idx]<=maxWeight)

            {

                return value[idx];

            }

            return 0;

        }

        if(dp[idx][maxWeight]!=-1) return dp[idx][maxWeight];

        int notPick = memo(idx-1,maxWeight,weight,value,dp);

        int pick=Integer.MIN\_VALUE;

        if(weight[idx]<=maxWeight)

        {

            pick=value[idx]+memo(idx-1,maxWeight-weight[idx],weight,value,dp);

        }

        return dp[idx][maxWeight]=Math.max(pick,notPick);

    }

    static int tab(int[] weight,int[] value,int n,int maxWeight,int[][] dp)

    {

        for(int i=0;i<=maxWeight;i++)

        {

            if(weight[0]<=i)

            {

                dp[0][i]=value[0];

            }

            else{

                dp[0][i]=0;

            }

        }

        for(int i=1;i<n;i++)

        {

            for(int wt=0;wt<=maxWeight;wt++)

            {

                int notPick=dp[i-1][wt];

                int pick=Integer.MIN\_VALUE;

                if(weight[i]<=wt)

                {

                    pick=value[i]+dp[i-1][wt-weight[i]];

                }

                dp[i][wt]=Math.max(pick, notPick);

            }

        }

        return dp[n-1][maxWeight];

    }

    static int knapsack(int[] weight, int[] value, int n, int maxWeight) {

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

        for(int[] i:dp)

        {

            Arrays.fill(i,-1);

        }

        // int ans=memo(n-1,maxWeight,weight,value,dp);

        return tab(weight, value, n, maxWeight, dp);

    }

}

**unbounded knapsack (infinte) : (Rod cutting)**

**Memoization :**

static int knapsackUtil(int[] wt, int[] val, int ind, int W, int[][] dp) {

*// Base case: If there are no more items to consider*

if (ind == 0) {

*// Calculate and return the maximum value possible*

return ((int) (W / wt[0])) \* val[0];

}

*// If the result for this subproblem has already been calculated, return it*

if (dp[ind][W] != -1)

return dp[ind][W];

*// Calculate the maximum value when the current item is not taken*

int notTaken = 0 + knapsackUtil(wt, val, ind - 1, W, dp);

*// Initialize the maximum value when the current item is taken as the minimum integer value*

int taken = Integer.MIN\_VALUE;

*// If the weight of the current item is less than or equal to the available capacity (W),*

*// calculate the maximum value when the current item is taken*

if (wt[ind] <= W)

taken = val[ind] + knapsackUtil(wt, val, ind, W - wt[ind], dp);

*// Store the result in the dp array and return it*

return dp[ind][W] = Math.max(notTaken, taken);

}

*// Function to find the maximum value of items that the thief can steal*

static int unboundedKnapsack(int n, int W, int[] val, int[] wt) {

*// Create a 2D array to store results of subproblems*

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

*// Initialize the dp array with -1 to indicate that subproblems are not solved yet*

for (int row[] : dp)

Arrays.fill(row, -1);

*// Call the knapsackUtil function to solve the problem*

return knapsackUtil(wt, val, n - 1, W, dp);

}

**Tabulation :**

static int unboundedKnapsack(int n, int W, int[] val, int[] wt) {

*// Create a 2D array to store results of subproblems*

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

*// Base condition: Initialize the dp array for the first item*

for (int i = wt[0]; i <= W; i++) {

dp[0][i] = ((int) i / wt[0]) \* val[0];

}

*// Fill the dp array using dynamic programming*

for (int ind = 1; ind < n; ind++) {

for (int cap = 0; cap <= W; cap++) {

*// Calculate the maximum value when the current item is not taken*

int notTaken = 0 + dp[ind - 1][cap];

*// Initialize the maximum value when the current item is taken as the minimum integer value*

int taken = Integer.MIN\_VALUE;

*// If the weight of the current item is less than or equal to the current capacity (cap),*

*// calculate the maximum value when the current item is taken*

if (wt[ind] <= cap)

taken = val[ind] + dp[ind][cap - wt[ind]];

*// Store the result in the dp array*

dp[ind][cap] = Math.max(notTaken, taken);

}

}

return dp[n - 1][W]; *// Return the maximum value that can be obtained*

}

**22) Coin Change (Leetcode - 322)**

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Example 1:

Input: coins = [1,2,5], amount = 11

Output: 3

Explanation: 11 = 5 + 5 + 1

Example 2:

Input: coins = [2], amount = 3

Output: -1

Example 3:

Input: coins = [1], amount = 0

Output: 0

Constraints:

1 <= coins.length <= 12

1 <= coins[i] <= 231 - 1

0 <= amount <= 104

**Solution :**

class Solution {

    public int memo(int idx,int[] coins,int amount,int[][] dp)

    {

        if(idx==0)

        {

            //coins=[4]  amount=8 return 2

            if(amount%coins[idx]==0)

            {

                return dp[idx][amount]=amount/coins[idx];

            }

            return (int) Math.pow(10,9);

        }

        if(dp[idx][amount]!=-1) return dp[idx][amount];

        int pick=Integer.MAX\_VALUE;

        if(coins[idx]<=amount)

        {

            pick=1+memo(idx,coins,amount-coins[idx],dp);

        }

        int notPick=memo(idx-1,coins,amount,dp);

        return dp[idx][amount]=Math.min(pick,notPick);

    }

    public int tab(int[] coins,int amount,int[][] dp)

    {

        for(int i=0;i<=amount;i++)

        {

            if(i%coins[0]==0)

            {

                dp[0][i]=i/coins[0];

            }

            else

            {

                dp[0][i]=(int) Math.pow(10,9);

            }

        }

        for(int i=1;i<coins.length;i++)

        {

            for(int amt=0;amt<=amount;amt++)

            {

                int notPick=dp[i-1][amt];

                int pick=Integer.MAX\_VALUE;

                if(coins[i]<=amt)

                {

                    pick=1+dp[i][amt-coins[i]];

                }

                dp[i][amt]=Math.min(pick,notPick);

            }

        }

        return dp[coins.length-1][amount];

    }

    public int coinChange(int[] coins, int amount)

    {

        int dp[][] =new int[coins.length][amount+1];

        for(int[] i:dp)

        {

            Arrays.fill(i,-1);

        }

        // int ans=memo(coins.length-1,coins,amount,dp);

        // return (ans>=Math.pow(10,9))?-1:ans;

        int ans=tab(coins,amount,dp);

        return (ans>=Math.pow(10,9))?-1:ans;

    }

}

**23) Target Sum (Leetcode - 494) //same as 20**

You are given an integer array nums and an integer target.

You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.

For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1".

Return the number of different expressions that you can build, which evaluates to target.

Example 1:

Input: nums = [1,1,1,1,1], target = 3

Output: 5

Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3.

-1 + 1 + 1 + 1 + 1 = 3

+1 - 1 + 1 + 1 + 1 = 3

+1 + 1 - 1 + 1 + 1 = 3

+1 + 1 + 1 - 1 + 1 = 3

+1 + 1 + 1 + 1 - 1 = 3

Example 2:

Input: nums = [1], target = 1

Output: 1

Constraints:

1 <= nums.length <= 20

0 <= nums[i] <= 1000

0 <= sum(nums[i]) <= 1000

-1000 <= target <= 1000

**Solution :**

class Solution {

    public int tab(int[] nums,int target)

    {

        int dp[][] = new int[nums.length][target+1];

        if(nums[0]==0) dp[0][0]=2;

        else dp[0][0]=1;

        if(nums[0]!=0 && nums[0]<=target) dp[0][nums[0]]=1;

        for(int i=1;i<nums.length;i++)

        {

            for(int tar=0;tar<=target;tar++)

            {

                int notPick=dp[i-1][tar];

                int pick = 0;

                if(tar-nums[i]>=0)

                {

                    pick=dp[i-1][tar-nums[i]];

                }

                dp[i][tar]=pick+notPick;

            }

        }

        return dp[nums.length-1][target];

    }

    public int findTargetSumWays(int[] nums, int target) {

        int totSum = 0;

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

        totSum += nums[i];

        }

        if(totSum-target <0 || (totSum-target)%2==1 ) return 0;

        return tab(nums,(totSum-target)/2);

}

}

**24) Coin Change II (Leetcode-518)**

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the number of combinations that make up that amount. If that amount of money cannot be made up by any combination of the coins, return 0.

You may assume that you have an infinite number of each kind of coin.

The answer is guaranteed to fit into a signed 32-bit integer.

Example 1:

Input: amount = 5, coins = [1,2,5]

Output: 4

Explanation: there are four ways to make up the amount:

5=5

5=2+2+1

5=2+1+1+1

5=1+1+1+1+1

Example 2:

Input: amount = 3, coins = [2]

Output: 0

Explanation: the amount of 3 cannot be made up just with coins of 2.

Example 3:

Input: amount = 10, coins = [10]

Output: 1

Constraints:

1 <= coins.length <= 300

1 <= coins[i] <= 5000

All the values of coins are unique.

0 <= amount <= 5000

**Solution :**

class Solution {

        public int tab(int[] coins,int amount,int[][] dp)

    {

        for(int i=0;i<=amount;i++)

        {

            if(i%coins[0]==0)

            {

                dp[0][i]=1;

            }

            else

            {

                dp[0][i]=0;

            }

        }

        for(int i=1;i<coins.length;i++)

        {

            for(int amt=0;amt<=amount;amt++)

            {

                int notPick=dp[i-1][amt];

                int pick=0;

                if(coins[i]<=amt)

                {

                    pick=dp[i][amt-coins[i]];

                }

                dp[i][amt]=pick+notPick;

            }

        }

        return dp[coins.length-1][amount];

    }

    public int change(int amount, int[] coins) {

        int dp[][] =new int[coins.length][amount+1];

        int ans=tab(coins,amount,dp);

        return ans;

    }

}

**25) Range Sum Query 2D - Immutable (Leetcode – 304)**

Given a 2D matrix matrix, handle multiple queries of the following type:

Calculate the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

Implement the NumMatrix class:

NumMatrix(int[][] matrix) Initializes the object with the integer matrix matrix.

int sumRegion(int row1, int col1, int row2, int col2) Returns the sum of the elements of matrix inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

You must design an algorithm where sumRegion works on O(1) time complexity.

Example 1:



Input

["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]

[[[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]

Output

[null, 8, 11, 12]

Explanation

NumMatrix numMatrix = new NumMatrix([[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]);

numMatrix.sumRegion(2, 1, 4, 3); // return 8 (i.e sum of the red rectangle)

numMatrix.sumRegion(1, 1, 2, 2); // return 11 (i.e sum of the green rectangle)

numMatrix.sumRegion(1, 2, 2, 4); // return 12 (i.e sum of the blue rectangle)

Constraints:

m == matrix.length

n == matrix[i].length

1 <= m, n <= 200

-104 <= matrix[i][j] <= 104

0 <= row1 <= row2 < m

0 <= col1 <= col2 < n

At most 104 calls will be made to sumRegion.

**Solution :**

class NumMatrix {

    int[][] dp;

    public NumMatrix(int[][] matrix) {

        this.dp=matrix;

        preCompute();

    }

    public void preCompute()

    {

        //dp[i][j]= sum from (0,0) - (i,j)

        for(int i=0;i<dp.length;i++)

        {

            for(int j=0;j<dp[0].length;j++)

            {

              if(i-1>=0)

              {

                dp[i][j]+=dp[i-1][j];

              }

              if(j-1>=0)

              {

                dp[i][j]+=dp[i][j-1];

              }

              if(i-1>=0 && j-1>=0) //dp[i-1][j-1] adding twice so remove once

              {

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

              }

            }

        }

    }

    public int sumRegion(int row1, int col1, int row2, int col2) {

        int ans=dp[row2][col2];

        if(col1-1>=0)

        {

            ans=ans-dp[row2][col1-1];

        }

        if(row1-1>=0)

        {

            ans=ans-dp[row1-1][col2];

        }

        if(row1-1>=0 && col1-1>=0)

        {

            ans=ans+dp[row1-1][col1-1];

        }

        return ans;

    }

}

**26) Assign Cookies (Leetcode - 455)(Greedy)**

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.

Each child i has a greed factor g[i], which is the minimum size of a cookie that the child will be content with; and each cookie j has a size s[j]. If s[j] >= g[i], we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number.

Example 1:

Input: g = [1,2,3], s = [1,1]

Output: 1

Explanation: You have 3 children and 2 cookies. The greed factors of 3 children are 1, 2, 3.

And even though you have 2 cookies, since their size is both 1, you could only make the child whose greed factor is 1 content.

You need to output 1.

Example 2:

Input: g = [1,2], s = [1,2,3]

Output: 2

Explanation: You have 2 children and 3 cookies. The greed factors of 2 children are 1, 2.

You have 3 cookies and their sizes are big enough to gratify all of the children,

You need to output 2.

Constraints:

1 <= g.length <= 3 \* 104

0 <= s.length <= 3 \* 104

1 <= g[i], s[j] <= 231 - 1

**Solution :**

class Solution {

    public int findContentChildren(int[] g, int[] s) {

        Arrays.sort(g);//sort children greed

        Arrays.sort(s);//sort cookies

        int l=0;

        int r=0;

        if(s.length==0)return 0;

        while(l<s.length && r<g.length)

        {

            if(s[l]>=g[r])

            {

                r+=1;

            }

            l+=1;

        }

        return r;

    }

}

**27) Longest Common Subsequence (Leetcode - 1143)**

Given two strings text1 and text2, return the length of their longest common subsequence. If there is no common subsequence, return 0.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde".

A common subsequence of two strings is a subsequence that is common to both strings.

Example 1:

Input: text1 = "abcde", text2 = "ace"

Output: 3

Explanation: The longest common subsequence is "ace" and its length is 3.

Example 2:

Input: text1 = "abc", text2 = "abc"

Output: 3

Explanation: The longest common subsequence is "abc" and its length is 3.

Example 3:

Input: text1 = "abc", text2 = "def"

Output: 0

Explanation: There is no such common subsequence, so the result is 0.

Constraints:

1 <= text1.length, text2.length <= 1000

text1 and text2 consist of only lowercase English characters.

**Solution :**

class Solution {

    public int memo(int i,int j,String text1,String text2,int[][] dp)

    {

        if(i<0 || j<0)

        {

            return 0;

        }

        if(dp[i][j]!=-1) return dp[i][j];

        if(text1.charAt(i)==text2.charAt(j))

        {

            return dp[i][j]=1+memo(i-1,j-1,text1,text2,dp);

        }

        return dp[i][j]=Math.max(memo(i-1,j,text1,text2,dp),memo(i,j-1,text1,text2,dp));

    }

    public int tab(String text1,String text2)

    {

        int dp[][] =new int[text1.length()+1][text2.length()+1];

        for(int i=1;i<=text1.length();i++)

        {

            for(int j=1;j<=text2.length();j++)

            {

                if(text1.charAt(i-1)==text2.charAt(j-1))

                {

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

                }

                else

                {

                    dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

                }

            }

        }

        return dp[text1.length()][text2.length()];

    }

    public int longestCommonSubsequence(String text1, String text2) {

        // int[][] dp = new int[text1.length()][text2.length()];

        // for(int i[]:dp)

        // {

        //     Arrays.fill(i,-1);

        // }

        // return memo(text1.length()-1,text2.length()-1,text1,text2,dp);

        return  tab(text1,text2);

    }

}

**28) Edit Distance (Leetcode - 72)**

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

You have the following three operations permitted on a word:

Insert a character

Delete a character

Replace a character

Example 1:

Input: word1 = "horse", word2 = "ros"

Output: 3

Explanation:

horse -> rorse (replace 'h' with 'r')

rorse -> rose (remove 'r')

rose -> ros (remove 'e')

Example 2:

Input: word1 = "intention", word2 = "execution"

Output: 5

Explanation:

intention -> inention (remove 't')

inention -> enention (replace 'i' with 'e')

enention -> exention (replace 'n' with 'x')

exention -> exection (replace 'n' with 'c')

exection -> execution (insert 'u')

Constraints:

0 <= word1.length, word2.length <= 500

word1 and word2 consist of lowercase English letters.

**Solution :**

class Solution {

    public int memo(int i,int j,String word1,String word2,int[][] dp)

    {

        if(i<0)

        {

            return j+1;

        }

        if(j<0)

        {

            return i+1;

        }

        if(dp[i][j]!=-1)

        {

            return dp[i][j];

        }

        if(word1.charAt(i)==word2.charAt(j))

        {

            return dp[i][j]=memo(i-1,j-1,word1,word2,dp);

        }

        return dp[i][j]=1+Math.min(Math.min(memo(i-1,j,word1,word2,dp),memo(i,j-1,word1,word2,dp)),memo(i-1,j-1,word1,word2,dp));

    }

    public int tab(String word1,String word2)

    {

        int n=word1.length(),m=word2.length();

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

        for(int i=1;i<=m;i++)

        {

            dp[0][i]=i;

        }

        for(int i=1;i<=n;i++)

        {

            dp[i][0]=i;

        }

        for(int i=1;i<=n;i++)

        {

            for(int j=1;j<=m;j++)

            {

                if(word1.charAt(i-1)==word2.charAt(j-1))

                {

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

                }

                else

                {

                    dp[i][j]=1+Math.min(Math.min(dp[i-1][j],dp[i][j-1]),dp[i-1][j-1]);

                }

            }

        }

        return dp[n][m];

    }

    public int minDistance(String word1, String word2) {

        // int n=word1.length(),m=word2.length();

        // int dp[][] = new int[n][m];

        // for(int[] i:dp)

        // {

        //     Arrays.fill(i,-1);

        // }

        // return memo(n-1,m-1,word1,word2,dp);

        return tab(word1,word2);

    }

}

**29) Longest Palindromic Subsequence (Leetcode - 516)**

Given a string s, find the longest palindromic subsequence's length in s.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: s = "bbbab"

Output: 4

Explanation: One possible longest palindromic subsequence is "bbbb".

Example 2:

Input: s = "cbbd"

Output: 2

Explanation: One possible longest palindromic subsequence is "bb".

Constraints:

1 <= s.length <= 1000

s consists only of lowercase English letters.

**Solution : (using LCS)**

class Solution {

    //longest common subSequence

    public int tab(String text1,String text2)

    {

        int dp[][] =new int[text1.length()+1][text2.length()+1];

        for(int i=1;i<=text1.length();i++)

        {

            for(int j=1;j<=text2.length();j++)

            {

                if(text1.charAt(i-1)==text2.charAt(j-1))

                {

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

                }

                else

                {

                    dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

                }

            }

        }

        return dp[text1.length()][text2.length()];

    }

    public int longestPalindromeSubseq(String s) {

        //longest common subSequence of string s and reverse of s is longest palindromic subsequence

        String rev = "";

        for(char c:s.toCharArray())

        {

            rev=Character.toString(c)+rev;

        }

        return tab(s,rev);

    }

}

**30) Minimum Insertion Steps to Make a String Palindrome (Leetcode - 1312)**

Given a string s. In one step you can insert any character at any index of the string.

Return the minimum number of steps to make s palindrome.

A Palindrome String is one that reads the same backward as well as forward.

Example 1:

Input: s = "zzazz"

Output: 0

Explanation: The string "zzazz" is already palindrome we do not need any insertions.

Example 2:

Input: s = "mbadm"

Output: 2

Explanation: String can be "mbdadbm" or "mdbabdm".

Example 3:

Input: s = "leetcode"

Output: 5

Explanation: Inserting 5 characters the string becomes "leetcodocteel".

Constraints:

1 <= s.length <= 500

s consists of lowercase English letters

**Solution : (Using LCS AND LPS)**

class Solution {

//LCS

    public int tab(String text1,String text2)

    {

        int dp[][] =new int[text1.length()+1][text2.length()+1];

        for(int i=1;i<=text1.length();i++)

        {

            for(int j=1;j<=text2.length();j++)

            {

                if(text1.charAt(i-1)==text2.charAt(j-1))

                {

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

                }

                else

                {

                    dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

                }

            }

        }

        return dp[text1.length()][text2.length()];

    }

    public int minInsertions(String s) {

        String rev = "";

        for(char c:s.toCharArray())

        {

            rev=Character.toString(c)+rev;

        }

        int lcs=tab(s,rev);

//LCS string and it reverse is LPS

//answer is string length – longest palindromic subsequence;

        return s.length()-lcs;

    }

}

**31) Delete Operation for Two Strings (Leetcode - 583)**

Given two strings word1 and word2, return the minimum number of steps required to make word1 and word2 the same.

In one step, you can delete exactly one character in either string.

Example 1:

Input: word1 = "sea", word2 = "eat"

Output: 2

Explanation: You need one step to make "sea" to "ea" and another step to make "eat" to "ea".

Example 2:

Input: word1 = "leetcode", word2 = "etco"

Output: 4

Constraints:

1 <= word1.length, word2.length <= 500

word1 and word2 consist of only lowercase English letters.

**Solution :** (LCS)

class Solution {

    //Longest Common SubSequence

    public int tab(String text1,String text2)

    {

        int dp[][] =new int[text1.length()+1][text2.length()+1];

        for(int i=1;i<=text1.length();i++)

        {

            for(int j=1;j<=text2.length();j++)

            {

                if(text1.charAt(i-1)==text2.charAt(j-1))

                {

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

                }

                else

                {

                    dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

                }

            }

        }

        return dp[text1.length()][text2.length()];

    }

    public int minDistance(String word1, String word2) {

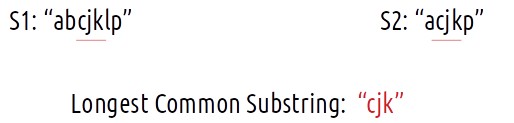
        int lcs=tab(word1,word2);

        return word1.length()+word2.length()-2\*lcs;

    }

}

**32) Longest Common Substring**



**Solution :**

static int lcs(String s1, String s2) {

int n = s1.length();

int m = s2.length();

*// Create a 2D array to store LCS lengths*

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

int ans = 0; *// Initialize a variable to store the maximum LCS length*

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= m; j++) {

*// If the characters at the current indices are the same, extend the LCS*

if (s1.charAt(i - 1) == s2.charAt(j - 1)) {

int val = 1 + dp[i - 1][j - 1];

dp[i][j] = val;

ans = Math.max(ans, val); *// Update the maximum LCS length*

} else {

dp[i][j] = 0; *// Reset LCS length if characters don't match*

}

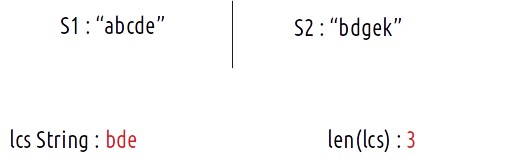
}

}

return ans; *// Return the length of the Longest Common Substring (LCS)*

}

**33) Print Longest Common Subsequence**



**Solution :**

static void lcs(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] = 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]);

}

}

int len=dp[n][m];

int i=n;

int j=m;

int index = len-1;

String str="";

for(int k=1; k<=len;k++){

str +="$"; *// dummy string*

}

StringBuilder ss= new StringBuilder(s1);

StringBuilder str2=new StringBuilder(str);

while(i>0 && j>0){

if(ss.charAt(i-1) == s2.charAt(j-1)){

str2.setCharAt(index,ss.charAt(i-1) );

index--;

i--;

j--;

}

else if(ss.charAt(i-1)>s2.charAt(j-1)){

i--;

}

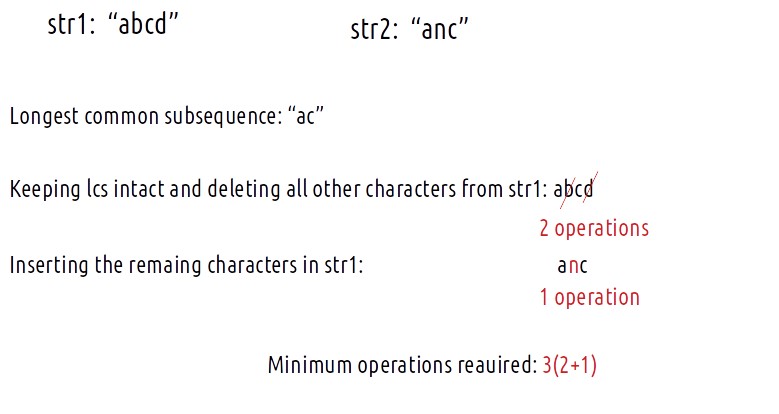
else j--;

}

System.out.println(str2);

}

**34)Minimum Insertions/Deletions to Convert String**



**Solution : (LCS)**

static int lcs(String s1, String s2) {

int n = s1.length();

int m = s2.length();

*// Create a 2D array to store the LCS lengths*

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

*// Initialize the dp array with -1*

for (int rows[] : dp)

Arrays.fill(rows, -1);

*// Fill the dp array using a bottom-up approach*

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] = Math.max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);

}

}

return dp[n][m];

}

*// Function to find the minimum operations required to convert str1 to str2*

static int canYouMake(String str1, String str2) {

int n = str1.length();

int m = str2.length();

*// Find the length of the LCS between str1 and str2*

int k = lcs(str1, str2);

*// The minimum operations required is the sum of the lengths of str1 and str2 minus twice the length of LCS*

return (n - k) + (m - k);

}

**35) Shortest Common Supersequence (Leetcode - 1092)**

Given two strings str1 and str2, return the shortest string that has both str1 and str2 as subsequences. If there are multiple valid strings, return any of them.

A string s is a subsequence of string t if deleting some number of characters from t (possibly 0) results in the string s.

Example 1:

Input: str1 = "abac", str2 = "cab"

Output: "cabac"

Explanation:

str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".

str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".

The answer provided is the shortest such string that satisfies these properties.

Example 2:

Input: str1 = "aaaaaaaa", str2 = "aaaaaaaa"

Output: "aaaaaaaa"

Constraints:

1 <= str1.length, str2.length <= 1000

str1 and str2 consist of lowercase English letters.

**Solution :**

class Solution {

    //longest Common SubSequence

    public String tab(String text1,String text2)

    {

        int dp[][] =new int[text1.length()+1][text2.length()+1];

        for(int i=1;i<=text1.length();i++)

        {

            for(int j=1;j<=text2.length();j++)

            {

                if(text1.charAt(i-1)==text2.charAt(j-1))

                {

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

                }

                else

                {

                    dp[i][j]=Math.max(dp[i-1][j],dp[i][j-1]);

                }

            }

        }

        String scs="";

        int i=text1.length(),j=text2.length();

        while(i>0 && j>0)

        {

            if(text1.charAt(i-1)==text2.charAt(j-1))

            {

                scs=Character.toString(text1.charAt(i-1))+scs;

                i=i-1;

                j=j-1;

            }

            else if(dp[i-1][j]>dp[i][j-1])

            {

               i=i-1;

               scs=Character.toString(text1.charAt(i))+scs;

            }

            else

            {

                j=j-1;

                scs=Character.toString(text2.charAt(j))+scs;

            }

        }

        while(i>0)

        {

            scs=text1.charAt(i-1)+scs;

            i--;

        }

        while(j>0)

        {

            scs=text2.charAt(j-1)+scs;

            j--;

        }

        return scs;

    }

    public String shortestCommonSupersequence(String str1, String str2) {

        return tab(str1,str2);

    }

}

**36) Distinct Subsequences (Leetcode - 115)**

Given two strings s and t, return the number of distinct subsequences of s which equals t.

The test cases are generated so that the answer fits on a 32-bit signed integer.

Example 1:

Input: s = "rabbbit", t = "rabbit"

Output: 3

Explanation:

As shown below, there are 3 ways you can generate "rabbit" from s.

rabbbit

rabbbit

rabbbit

Example 2:

Input: s = "babgbag", t = "bag"

Output: 5

Explanation:

As shown below, there are 5 ways you can generate "bag" from s.

babgbag

babgbag

babgbag

babgbag

babgbag

Constraints:

1 <= s.length, t.length <= 1000

s and t consist of English letters.

**Solution :**

class Solution {

    private static int distinct(int i,int j,String s,String t,int[][] dp) {

        if(j==-1)

        {

            return 1;

        }

        if(i==-1)return 0;

        if(dp[i][j]!=-1) return dp[i][j];

        if(s.charAt(i)==t.charAt(j))

        {

            return dp[i][j]=distinct(i-1,j-1,s,t,dp)+distinct(i-1,j,s,t,dp);

        }

        return dp[i][j]=distinct(i-1,j,s,t,dp);

    }

    public static int tab(String s,String t)

    {

        int[][] dp = new int[s.length()+1][t.length()+1];

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

        {

            dp[0][i]=0;

        }

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

        {

            dp[i][0]=1;

        }

        for(int i=1;i<=s.length();i++)

        {

            for(int j=1;j<=t.length();j++)

            {

                if(s.charAt(i-1)==t.charAt(j-1))

                {

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

                }

                else

                {

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

                }

            }

        }

         return dp[s.length()][t.length()];

    }

    public int numDistinct(String s, String t) {

        // int[][] dp=new int[s.length()][t.length()];

        // for(int i[]:dp)

        // {

        //     Arrays.fill(i,-1);

        // }

        // return distinct(s.length()-1,t.length()-1,s,t,dp);

        return tab(s,t);

    }

}

**37) Wildcard Matching (Leetcode - 44)**

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '\*' where:

'?' Matches any single character.

'\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

Example 1:

Input: s = "aa", p = "a"

Output: false

Explanation: "a" does not match the entire string "aa".

Example 2:

Input: s = "aa", p = "\*"

Output: true

Explanation: '\*' matches any sequence.

Example 3:

Input: s = "cb", p = "?a"

Output: false

Explanation: '?' matches 'c', but the second letter is 'a', which does not match 'b'.

Constraints:

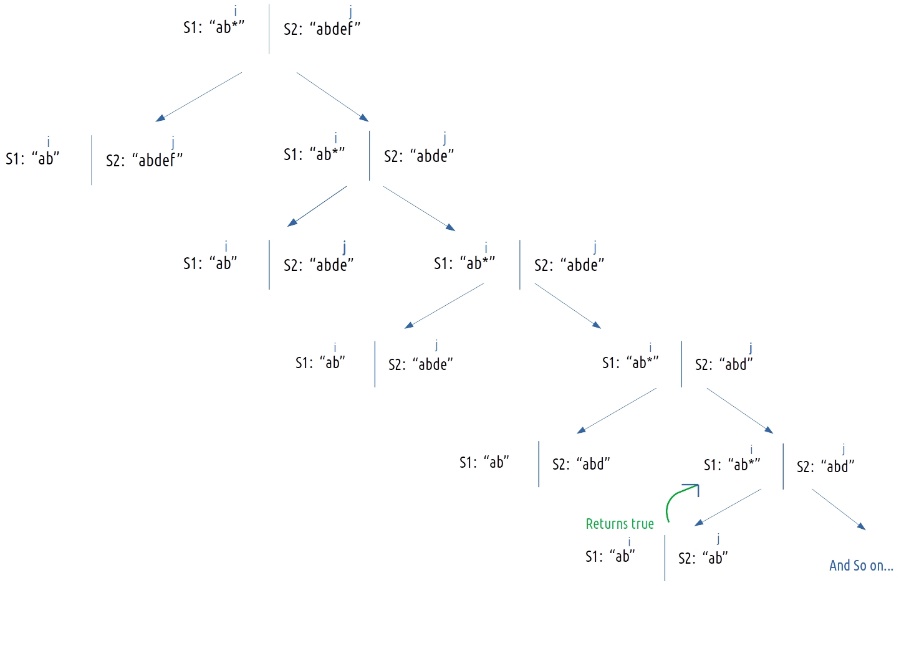
0 <= s.length, p.length <= 2000

s contains only lowercase English letters.

p contains only lowercase English letters, '?' or '\*'.

**Solution :**

**(‘\*’ case)**



class Solution {

    public boolean memo(int i,int j,String s,String p,int[][] dp)

    {

        //both get exhausted

        if(i<0 && j<0)

        {

            return true;

        }

         //p get exhausted

        if(i>=0 && j<0)

        {

            return false;

        }

        //s getExhausted

        //s="" p="\*\*"

        if(i<0 && j>=0)

        {

            for(int k=0;k<=j;k++)

            {

                if(p.charAt(k)!='\*')

                {

                    return false;

                }

            }

            return true;

        }

        if(dp[i][j]!=-1) return (dp[i][j]==1)?true:false;

        //match

        if(s.charAt(i)==p.charAt(j) || p.charAt(j)=='?')

        {

            return memo(i-1,j-1,s,p,dp);

        }

        if(p.charAt(j)=='\*')

        {

            //notMatch

            //p[j] is \* then consider it "" i.e memo(i,j-1,s,p)

            //another case \* to match the character and star is still there //memo(i-1,j,s,p)

            dp[i][j]=(memo(i, j-1, s, p, dp) || memo(i-1, j, s, p, dp)) ? 1 : 0;

            return (dp[i][j]==1)?true:false;

        }

        return false;

    }

    public boolean tab(String s,String p)

    {

        boolean[][] dp = new boolean[s.length()+1][p.length()+1];

        //both get exhausted

        dp[0][0]=true;

        //p get exhausted

        for(int i=1;i<=s.length();i++)

        {

            dp[i][0]=false;

        }

        //s get exhausted

        for(int j=1;j<=p.length();j++)

        {

            if(p.charAt(j-1)=='\*')

            {

                dp[0][j]=true;

            }

            else

                break;

        }

        for(int i=1;i<=s.length();i++)

        {

            for(int j=1;j<=p.length();j++)

            {

                if(s.charAt(i-1)==p.charAt(j-1)|| p.charAt(j-1)=='?')

                {

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

                }

                else if(p.charAt(j-1)=='\*')

                {

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

                }

                else

                {

                    dp[i][j]=false;

                }

            }

        }

        return dp[s.length()][p.length()];

    }

    public boolean isMatch(String s, String p) {

        // int dp[][] = new int[s.length()][p.length()];

        // for(int[] i:dp)

        // {

        //     Arrays.fill(i,-1);

        // }

        // return memo(s.length()-1,p.length()-1,s,p,dp);

        return tab(s,p);

        }

}

**38) Best Time to Buy and Sell Stock (Leetcode - 121 )**

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:

1 <= prices.length <= 105

0 <= prices[i] <= 104

**Solution :**

class Solution {

    public int maxProfit(int[] prices) {

        //finding min element in left of the array to that position

        int min=prices[0];

        int profit=0;

        for(int i:prices)

        {

            profit=Math.max(profit,i-min);

            min=Math.min(i,min);

        }

        return profit;

    }

}

**39) Best Time to Buy and Sell Stock II (Leetcode - 122)**

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return the maximum profit you can achieve.

Example 1:

Input: prices = [7,1,5,3,6,4]

Output: 7

Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

Total profit is 4 + 3 = 7.

Example 2:

Input: prices = [1,2,3,4,5]

Output: 4

Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Total profit is 4.

Example 3:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

Constraints:

1 <= prices.length <= 3 \* 104

0 <= prices[i] <= 104

**Solution :**

class Solution {

public int memo(int i,int buy,int[] prices,int[][] dp)

{

if(i==prices.length)

{

return 0;

}

if(dp[i][buy]!=-1)

{

return dp[i][buy];

}

int profit=0;

//buy the stock

if(buy==1)

{

//buy the stock i.e -prices[i]+memo(i+1,0,p) i.e 5-7=-2 3-7=0 (7 is deleting)

//skip to buy the present stock memo(i+1,1,p)

profit=Math.max(-prices[i]+memo(i+1,0,prices,dp),memo(i+1,1,prices,dp));

}

//sell

if(buy==0)

{

//sell the stock i.e prices[i]+memo(i+1,1,p)

//skip to sell the curretn stock memo(i+1,0,p)

profit=Math.max(prices[i]+memo(i+1,1,prices,dp),memo(i+1,0,prices,dp));

}

return dp[i][buy]=profit;

}

public int tab(int prices[])

{

int[][] dp =new int[prices.length+1][2];

dp[prices.length][0]=0;

dp[prices.length][1]=0;

for(int i=prices.length-1;i>=0;i--)

{

for(int j=0;j<2;j++)

{

if(j==1)

{

dp[i][j]=Math.max(-prices[i]+dp[i+1][0],dp[i+1][1]);

}

if(j==0)

{

dp[i][j]=Math.max(prices[i]+dp[i+1][1],dp[i+1][0]);

}

}

}

return dp[0][1];

}

int space(int[] Arr, int n) {

// Create arrays 'ahead' and 'cur' to store the maximum profit ahead and current profit

int[] ahead = new int[2];

// Base condition: If we have no stocks to buy or sell, profit is 0

ahead[0] = ahead[1] = 0;

int profit = 0;

// Iterate through the array in reverse to calculate the maximum profit

for (int ind = n - 1; ind >= 0; ind--) {

int[] cur = new int[2];

for (int buy = 0; buy <= 1; buy++) {

if (buy == 1) { // We can buy the stock

profit = Math.max(0 + ahead[1], -Arr[ind] + ahead[0]);

}

if (buy == 0) { // We can sell the stock

profit = Math.max(0 + ahead[0], Arr[ind] + ahead[1]);

}

cur[buy] = profit;

}

// Update the 'ahead' array with the current profit values

ahead=cur;

}

return ahead[1]; // The maximum profit is stored in 'ahead[1]'

}

public int maxProfit(int[] prices) {

// int[][] dp=new int[prices.length][2];

// for(int[] i:dp)

// {

// Arrays.fill(i,-1);

// }

// return memo(0,1,prices,dp);

//return tab(prices);

return space(prices,prices.length);

}

}

**//Space solution can also written as**

int space(int[] Arr, int n) {

// Create arrays 'ahead' and 'cur' to store the maximum profit ahead and current profit

int[] ahead = new int[2];

// Base condition: If we have no stocks to buy or sell, profit is 0

ahead[0] = ahead[1] = 0;

int profit = 0;

// Iterate through the array in reverse to calculate the maximum profit

for (int ind = n - 1; ind >= 0; ind--) {

int[] cur = new int[2];

cur[1] = Math.max(0 + ahead[1], -Arr[ind] + ahead[0]);

cur[0] = Math.max(0 + ahead[0], Arr[ind] + ahead[1]);

// Update the 'ahead' array with the current profit values

ahead=cur;

}

return ahead[1]; // The maximum profit is stored in 'ahead[1]'

}

**//Greedy Solution**

class Solution {

public int maxProfit(int[] prices) {

int total = 0;

for (int i=0; i< prices.length-1; i++) {

if (prices[i+1]>prices[i]) total += prices[i+1]-prices[i];

}

return total;

}

}

**40) Best Time to Buy and Sell Stock III (Leetcode - 123)**

You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete at most two transactions.

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

Example 1:

Input: prices = [3,3,5,0,0,3,1,4]

Output: 6

Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

Example 2:

Input: prices = [1,2,3,4,5]

Output: 4

Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.

Example 3:

Input: prices = [7,6,4,3,1]

Output: 0

Explanation: In this case, no transaction is done, i.e. max profit = 0.

Constraints:

1 <= prices.length <= 105

0 <= prices[i] <= 105

**Solution :**

class Solution {

    public int memo(int i,int buy,int cap,int[] prices,int[][][] dp)

    {

        if(i==prices.length) return 0;

        if(cap==0) return 0;

        if(dp[i][buy][cap]!=-1) return dp[i][buy][cap];

        int profit=0;

        if(buy==1)

        {

            profit =  Math.max(-prices[i]+memo(i+1,0,cap,prices,dp),memo(i+1,1,cap,prices,dp));

        }

        if(buy==0)

        {

            //capacity is reduced if total buy and sell is done

            profit =  Math.max(prices[i]+memo(i+1,1,cap-1,prices,dp),memo(i+1,0,cap,prices,dp));

        }

        return dp[i][buy][cap]=profit;

    }

    public int tab(int[] prices)

    {

         int dp[][][] = new int[prices.length+1][2][3];

         for(int i=prices.length-1;i>=0;i--)

         {

            for(int buy=0;buy<=1;buy++)

            {

                for(int cap=1;cap<=2;cap++)

                {

                    if(buy==1)

                    {

                        dp[i][buy][cap]=Math.max(-prices[i]+dp[i+1][0][cap],dp[i+1][1][cap]);

                    }

                    if(buy==0)

                    {

                        dp[i][buy][cap]=Math.max(prices[i]+dp[i+1][1][cap-1],dp[i+1][0][cap]);

                    }

                }

            }

         }

         return dp[0][1][2];

    }

    public int maxProfit(int[] prices) {

        // int dp[][][] = new int[prices.length][2][3];

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

        // {

        //     for(int j[]:dp[i])

        //     {

        //         Arrays.fill(j,-1);

        //     }

        // }

        // //cap atmost = 2;

        // return memo(0,1,2,prices,dp);

        return tab(prices);

    }

}

**//space optimization**

static int maxProfit(int[] prices) {

int n = prices.length;

*// Create a 2D array 'ahead' and 'cur' to store profit values*

int[][] ahead = new int[2][3];

int[][] cur = new int[2][3];

*// Loop through the prices array, starting from the second last stock (ind=n-1)*

for (int ind = n - 1; ind >= 0; ind--) {

for (int buy = 0; buy <= 1; buy++) {

for (int cap = 1; cap <= 2; cap++) {

if (buy == 0) { *// We can buy the stock*

cur[buy][cap] = Math.max(0 + ahead[0][cap],

-prices[ind] + ahead[1][cap]);

}

if (buy == 1) { *// We can sell the stock*

cur[buy][cap] = Math.max(0 + ahead[1][cap],

prices[ind] + ahead[0][cap - 1]);

}

}

}

*// Update 'ahead' with the values in 'cur'*

for (int i = 0; i < 2; i++) {

for (int j = 1; j < 3; j++) {

ahead[i][j] = cur[i][j];

}

}

}

*// The maximum profit with 2 transactions is stored in ahead[0][2]*

return ahead[0][2];

}

**41) Best Time to Buy and Sell Stock IV(Leetcode - 188)**

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

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

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

Example 1:

Input: k = 2, prices = [2,4,1]

Output: 2

Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.

Example 2:

Input: k = 2, prices = [3,2,6,5,0,3]

Output: 7

Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Constraints:

1 <= k <= 100

1 <= prices.length <= 1000

0 <= prices[i] <= 1000

**Solution ://same as Best Time to Buy and Sell Stock III but cap = k**

class Solution {

    public int tab(int[] prices,int k)

    {

         int dp[][][] = new int[prices.length+1][2][k+1];

         for(int i=prices.length-1;i>=0;i--)

         {

            for(int buy=0;buy<=1;buy++)

            {

                for(int cap=1;cap<=k;cap++)

                {

                    if(buy==1)

                    {

                        dp[i][buy][cap]=Math.max(-prices[i]+dp[i+1][0][cap],dp[i+1][1][cap]);

                    }

                    if(buy==0)

                    {

                        dp[i][buy][cap]=Math.max(prices[i]+dp[i+1][1][cap-1],dp[i+1][0][cap]);

                    }

                }

            }

         }

         return dp[0][1][k];

    }

    public int maxProfit(int k, int[] prices) {

        return tab(prices,k);

    }

}

**42) Best Time to Buy and Sell Stock with Transaction Fee (Leetcode - 714)**

You are given an array prices where prices[i] is the price of a given stock on the ith day, and an integer fee representing a transaction fee.

Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

Note:

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

The transaction fee is only charged once for each stock purchase and sale.

Example 1:

Input: prices = [1,3,2,8,4,9], fee = 2

Output: 8

Explanation: The maximum profit can be achieved by:

- Buying at prices[0] = 1

- Selling at prices[3] = 8

- Buying at prices[4] = 4

- Selling at prices[5] = 9

The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

Example 2:

Input: prices = [1,3,7,5,10,3], fee = 3

Output: 6

Constraints:

1 <= prices.length <= 5 \* 104

1 <= prices[i] < 5 \* 104

0 <= fee < 5 \* 104

**Solution ://same as Best Time to Buy and Sell Stock II but remove fee for every transaction completed(transacton is said to be completed if a stock is buy and sell)**

class Solution {

    public int tab(int prices[],int fee)

    {

        int[][] dp =new int[prices.length+1][2];

        dp[prices.length][0]=0;

        dp[prices.length][1]=0;

        for(int i=prices.length-1;i>=0;i--)

        {

            for(int j=0;j<2;j++)

            {

                if(j==1)

                {

                    dp[i][j]=Math.max(-prices[i]+dp[i+1][0],dp[i+1][1]);

                }

                if(j==0)

                {

//remove fee

                    dp[i][j]=Math.max(prices[i]-fee+dp[i+1][1],dp[i+1][0]);

                }

            }

        }

        return dp[0][1];

    }

    public int maxProfit(int[] prices, int fee) {

        return tab(prices,fee);

    }

}

**43) Best Time to Buy and Sell Stock with Cooldown (Leetcode - 309)**

You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times) with the following restrictions:

After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).

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

Example 1:

Input: prices = [1,2,3,0,2]

Output: 3

Explanation: transactions = [buy, sell, cooldown, buy, sell]

Example 2:

Input: prices = [1]

Output: 0

Constraints:

1 <= prices.length <= 5000

0 <= prices[i] <= 1000

**Solution : (same as Best Time to Buy and Sell Stock II but when one transaction is completed then skip the next element )**

class Solution {

    public int tab(int prices[])

    {

        int[][] dp =new int[prices.length+2][2];

        dp[prices.length][0]=0;

        dp[prices.length][1]=0;

        for(int i=prices.length-1;i>=0;i--)

        {

            for(int j=0;j<2;j++)

            {

                if(j==1)

                {

                    dp[i][j]=Math.max(-prices[i]+dp[i+1][0],dp[i+1][1]);

                }

                if(j==0)

                {

//After selling dp[i+2][1]

                    dp[i][j]=Math.max(prices[i]+dp[i+2][1],dp[i+1][0]);

                }

            }

        }

        return dp[0][1];

    }

    public int maxProfit(int[] prices) {

        return tab(prices);

    }

}

**44) Largest Divisible Subset (Leetcode - 368)//LIS Variation**

Given a set of distinct positive integers nums, return the largest subset answer such that every pair (answer[i], answer[j]) of elements in this subset satisfies:

answer[i] % answer[j] == 0, or

answer[j] % answer[i] == 0

If there are multiple solutions, return any of them.

Example 1:

Input: nums = [1,2,3]

Output: [1,2]

Explanation: [1,3] is also accepted.

Example 2:

Input: nums = [1,2,4,8]

Output: [1,2,4,8]

Constraints:

1 <= nums.length <= 1000

1 <= nums[i] <= 2 \* 109

All the integers in nums are unique.

**Solution ://longest divisible subsequence**

**Sort the elements first.**

**The distinguishing factor between longest increasing subsequence and longest divisible subsequence is that we used to insert the element if arr[i] > arr[prev] but here we will insert the element when arr[i] % arr[prev] == 0.**

class Solution {

    public List<Integer> tab(int[] nums)

    {

        Arrays.sort(nums);

        int list[] = new int[nums.length];

        int prev[] = new int[nums.length];

        int maxInd=0,max=1;

        list[0]=1;

        for(int i=1;i<nums.length;i++)

        {

            prev[i]=i;

            for(int j=0;j<i;j++)

            {

                if(nums[i]%nums[j]==0 && list[i]<list[j])

                {

                    list[i]=list[j];

                    prev[i]=j;

                }

            }

            list[i]+=1;

            if(max<list[i])

            {

                max=list[i];

                maxInd=i;

            }

        }

        List<Integer> res = new ArrayList<>();

        res.add(nums[maxInd]);

        while(prev[maxInd]!=maxInd)

        {

            maxInd=prev[maxInd];

            res.add(nums[maxInd]);

        }

        return res;

    }

    public List<Integer> largestDivisibleSubset(int[] nums) {

        return tab(nums);

    }

}

**45) Longest String Chain (Leetcode - 1048)**

You are given an array of words where each word consists of lowercase English letters.

wordA is a predecessor of wordB if and only if we can insert exactly one letter anywhere in wordA without changing the order of the other characters to make it equal to wordB.

For example, "abc" is a predecessor of "abac", while "cba" is not a predecessor of "bcad".

A word chain is a sequence of words [word1, word2, ..., wordk] with k >= 1, where word1 is a predecessor of word2, word2 is a predecessor of word3, and so on. A single word is trivially a word chain with k == 1.

Return the length of the longest possible word chain with words chosen from the given list of words.

Example 1:

Input: words = ["a","b","ba","bca","bda","bdca"]

Output: 4

Explanation: One of the longest word chains is ["a","ba","bda","bdca"].

Example 2:

Input: words = ["xbc","pcxbcf","xb","cxbc","pcxbc"]

Output: 5

Explanation: All the words can be put in a word chain ["xb", "xbc", "cxbc", "pcxbc", "pcxbcf"].

Example 3:

Input: words = ["abcd","dbqca"]

Output: 1

Explanation: The trivial word chain ["abcd"] is one of the longest word chains.

["abcd","dbqca"] is not a valid word chain because the ordering of the letters is changed.

Constraints:

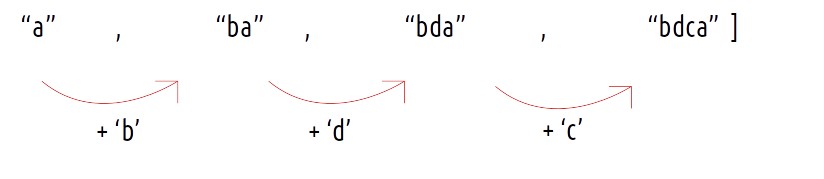
1 <= words.length <= 1000

1 <= words[i].length <= 16

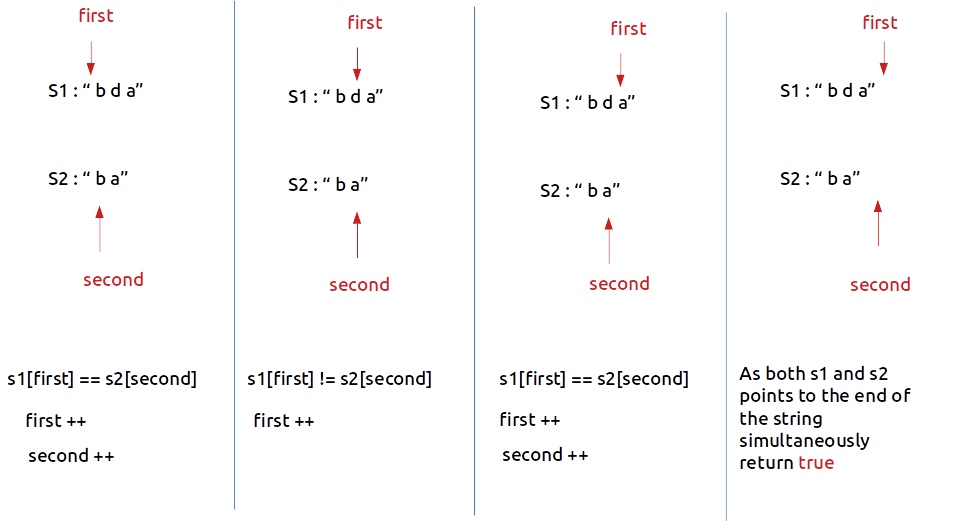
words[i] only consists of lowercase English letters.

**Solution :**

Two consecutive strings in this string chain need to have an insertion of a single character. The character can be added to any place on the string.



Compare(s1,s2)



class Solution {

    boolean compare(String s1,String s2)

    {

        if(s1.length()!=s2.length()+1)

        {

            return false;

        }

        int first=0;

        int second=0;

        while(first<s1.length())

        {

            if(second<s2.length() && s1.charAt(first)==s2.charAt(second))

            {

                first++;

                second++;

            }

            else

            {

                first++;

            }

        }

        if(first==s1.length() && second==s2.length())

        {

            return true;

        }

        return false;

    }

     int lis(String words[], int n)

    {

        int lis[] = new int[n];

        int i, j, max = 1,maxInd=0;

       lis[0]=1;

        for (i = 1; i < n; i++)

        {

            for (j = 0;j<i; j++)

            {

                if (compare(words[i],words[j]) && lis[i] < lis[j] )

                {

                    lis[i] = lis[j] ;

                }

            }

            lis[i]+=1;

            max=Math.max(max,lis[i]);

        }

        return max;

    }

    public int longestStrChain(String[] words) {

        Arrays.sort(words,(a,b)-> a.length()-b.length());

        return lis(words,words.length);

    }

}

**46) Minimum Number of Removals to Make Mountain Array(Leetcode - 1671)**

**//Longest bitonic series**

You may recall that an array arr is a mountain array if and only if:

arr.length >= 3

There exists some index i (0-indexed) with 0 < i < arr.length - 1 such that:

arr[0] < arr[1] < ... < arr[i - 1] < arr[i]

arr[i] > arr[i + 1] > ... > arr[arr.length - 1]

Given an integer array nums​​​, return the minimum number of elements to remove to make nums​​​ a mountain array.

Example 1:

Input: nums = [1,3,1]

Output: 0

Explanation: The array itself is a mountain array so we do not need to remove any elements.

Example 2:

Input: nums = [2,1,1,5,6,2,3,1]

Output: 3

Explanation: One solution is to remove the elements at indices 0, 1, and 5, making the array nums = [1,5,6,3,1].

Constraints:

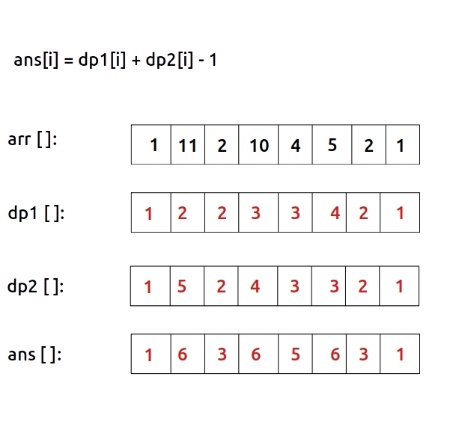
3 <= nums.length <= 1000

1 <= nums[i] <= 109

It is guaranteed that you can make a mountain array out of nums.

**Bitonic series**

Input: nums = [1, 11, 2, 10, 4, 5, 2, 1]  
Output: 6  
Explanation: The bitonic sequence  
{1, 2, 10, 4, 2, 1} has length 6. //dp1 == LIS and dp2 == LDS



Max(ans) is lbs.

**Solution :**

class Solution {

    //longest bitonic series

    static int lbs(int arr[], int n)

    {

        //LIS

        int lis[] = new int[n];

        int i, j;

       lis[0]=1;

        for (i = 1; i < n; i++)

        {

            for (j = 0;j<i; j++)

            {

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

                {

                    lis[i] = lis[j] ;

                }

            }

            lis[i]+=1;

        }

        //LDS

        int lds[] = new int[n];

        lds[n-1]=1;

        for (i = n-2;i >=0; i--)

        {

            for (j = n-1;j>i; j--)

            {

                if (arr[i] > arr[j] && lds[i] < lds[j] )

                {

                    lds[i] = lds[j] ;

                }

            }

            lds[i]+=1;

        }

        int lbs=0;

        for(i=0;i<n-1;i++)

        {

// lis[i]=1 or lds[i]=1 then it is not form a mountain

            if(lis[i]>1 && lds[i]>1)

            {

                //for every index the element is included in both lis and lds so remove 1 time

                lbs=Math.max(lis[i]+lds[i]-1,lbs);

            }

        }

        return lbs;

    }

    public int minimumMountainRemovals(int[] nums) {

        return nums.length-lbs(nums,nums.length);

    }

}

**//using binary search also we find LIS and LDS**

    public int lengthOfLIS(int[] nums) {

        List<Integer> lis = new ArrayList<>(nums.length);

        int[] lisA = new int[nums.length];

        int j=0;

        for (int n : nums) {

            int i = Collections.binarySearch(lis, n);

            //if element not there it returns -ve index of where ele should be present

            if (i < 0) i = -i - 1;

            if (i == lis.size())

                lis.add(n);

            else

                lis.set(i, n);

            lisA[j]=lis.size();

            j++;

        }

//printing lisA

        System.out.println(Arrays.toString(lisA));

        return lis.size();

    }

**47) Number of Longest Increasing Subsequence (Leetcode - 673)**

Given an integer array nums, return the number of longest increasing subsequences.

Notice that the sequence has to be strictly increasing.

Example 1:

Input: nums = [1,3,5,4,7]

Output: 2

Explanation: The two longest increasing subsequences are [1, 3, 4, 7] and [1, 3, 5, 7].

Example 2:

Input: nums = [2,2,2,2,2]

Output: 5

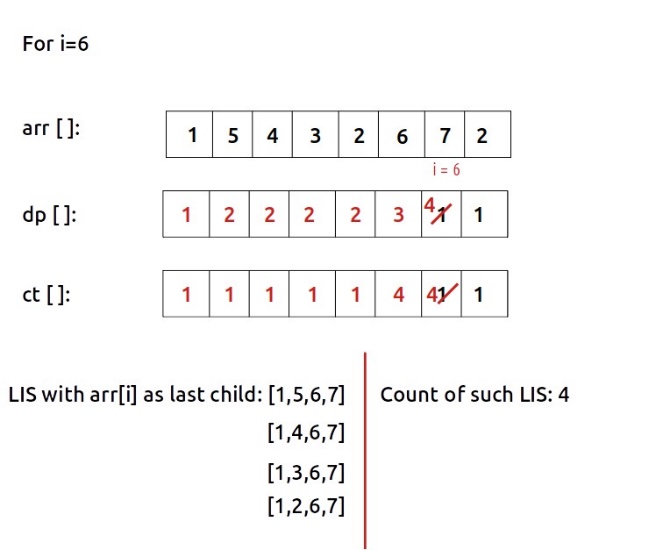
Explanation: The length of the longest increasing subsequence is 1, and there are 5 increasing subsequences of length 1, so output 5.

Constraints:

1 <= nums.length <= 2000

-106 <= nums[i] <= 106

Dp==LIS



**Solution :**

class Solution {

    static int lis(int arr[], int n)

    {

        int lis[] = new int[n];

        int count[] = new int[n];

        int i, j, max = 1,maxInd=0;

        lis[0]=1;

        Arrays.fill(count,1);

        for (i = 1; i < n; i++)

        {

            for (j = 0;j<i; j++)

            {

                if (arr[i] > arr[j])

                {

                    if(lis[i] < lis[j])

                    {

                        lis[i] = lis[j];

                        count[i] = count[j];

                    }

                     else if (lis[j] == lis[i]) {

                        count[i] += count[j];

                    }

                }

            }

            lis[i]++;

            max=Math.max(max,lis[i]);

             int result = 0;

         }

         int result=0;

         for (i = 0; i < n; i++) {

            if (lis[i] == max) {

                result += count[i];

            }

        }

        return result;

    }

    public int findNumberOfLIS(int[] nums) {

        return lis(nums,nums.length);

    }

}

**48) Matrix Chain Multiplication (Coding ninjas)**

Given a chain of matrices A1, A2, A3,.....An. Your task is to find out the minimum cost to multiply these matrices. The cost of matrix multiplication is defined as the number of scalar multiplications. A Chain of matrices A1, A2, A3,.....An is represented by a sequence of numbers in an array ‘arr’ where the dimension of 1st matrix is equal to arr[0] \* arr[1] , 2nd matrix is arr[1] \* arr[2], and so on.

For example:

For arr[ ] = { 10, 20, 30, 40}, matrix A1 = [10 \* 20], A2 = [20 \* 30], A3 = [30 \* 40]

Scalar multiplication of matrix with dimension 10 \* 20 is equal to 200.

Detailed explanation ( Input/output format, Notes, Images )

Sample Input 1:

2

4

4 5 3 2

4

10 15 20 25

Sample Output 1:

70

8000

Sample Output Explanation 1:

In the first test case, there are three matrices of dimensions A = [4 5], B = [5 3] and C = [3 2]. The most efficient order of multiplication is A \* ( B \* C).

Cost of ( B \* C ) = 5 \* 3 \* 2 = 30 and (B \* C) = [5 2] and A \* (B \* C) = [ 4 5] \* [5 2] = 4 \* 5 \* 2 = 40. So the overall cost is equal to 30 + 40 =70.

In the second test case, there are two ways to multiply the chain - A1\*(A2\*A3) or (A1\*A2)\*A3.

If we multiply in order- A1\*(A2\*A3), then the number of multiplications required is 11250.

If we multiply in order- (A1\*A2)\*A3, then the number of multiplications required is 8000.

Thus a minimum number of multiplications required is 8000.

Sample Input 2:

1

4

1 4 3 2

Sample Output 2:

18

Explanation of Sample Output 2:

In the first test case, there are three matrices of dimensions A = [1 4], B = [4 3] and C = [3 2]. The most efficient order of multiplication is (A \* B) \* C .

**Solution :**

import java.util.\* ;

import java.io.\*;

public class Solution {

    public static int cost(int i,int j,int[] arr,int[][] dp)

    {

        if(i==j) return 0;

        if(dp[i][j]!=-1) return dp[i][j];

        int min=Integer.MAX\_VALUE;

        for(int k=i;k<j;k++)

        {

            int cost=arr[i-1]\*arr[k]\*arr[j]+cost(i,k,arr,dp)+cost(k+1,j,arr,dp);

            min=Math.min(cost,min);

        }

        return dp[i][j]=min;

    }

    public static int tab(int[] arr)

    {

        int[][] dp = new int[arr.length][arr.length];

        for(int i=0;i<arr.length;i++)

        {

            dp[i][i]=0;

        }

        for(int i=arr.length-1;i>=1;i--)

        {

            //j should always starts after i

            for(int j=i+1;j<arr.length;j++)

            {

                int min=Integer.MAX\_VALUE;

                for(int k=i;k<j;k++)

                {

                    int cost=arr[i-1]\*arr[k]\*arr[j]+dp[i][k]+dp[k+1][j];

                    min=Math.min(cost,min);

                }

                dp[i][j]=min;

            }

        }

        return dp[1][arr.length-1];

    }

    public static int matrixMultiplication(int[] arr , int N) {

        // int dp[][] = new int[arr.length][arr.length];

        // for( int[] i:dp)

        // {

        //  Arrays.fill(i,-1);

        // }

        // return cost(1,arr.length-1,arr,dp);

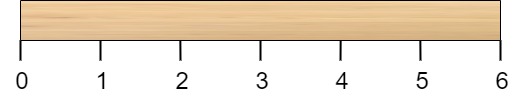
        return tab(arr);

    }

}

**49) Minimum Cost to Cut a Stick (Leetcode - 1547)**

Given a wooden stick of length n units. The stick is labelled from 0 to n. For example, a stick of length 6 is labelled as follows:



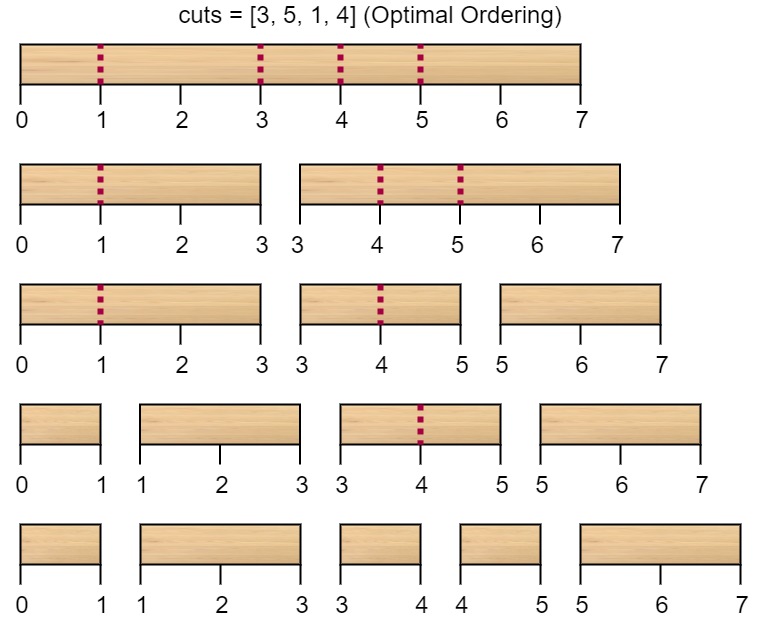
Given an integer array cuts where cuts[i] denotes a position you should perform a cut at.

You should perform the cuts in order, you can change the order of the cuts as you wish.

The cost of one cut is the length of the stick to be cut, the total cost is the sum of costs of all cuts. When you cut a stick, it will be split into two smaller sticks (i.e. the sum of their lengths is the length of the stick before the cut). Please refer to the first example for a better explanation.

Return the minimum total cost of the cuts.

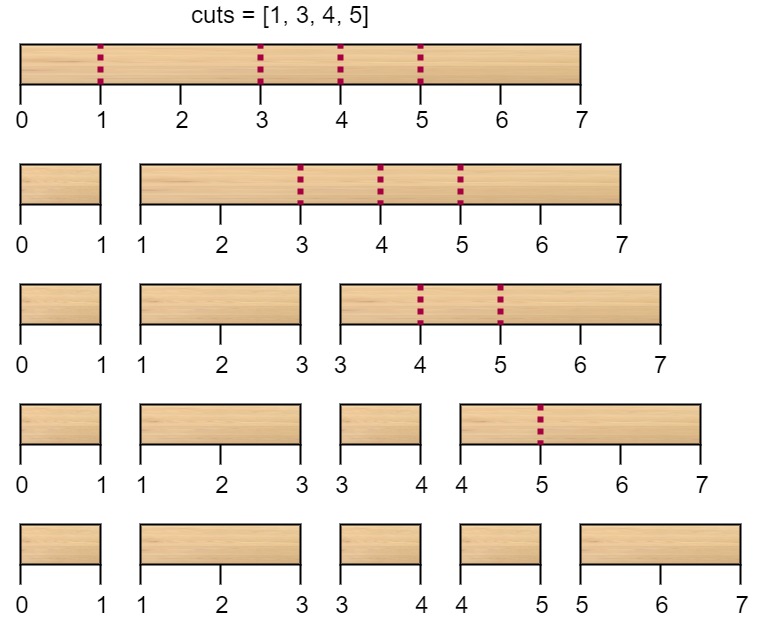
Example 1:



Input: n = 7, cuts = [1,3,4,5]

Output: 16

Explanation: Using cuts order = [1, 3, 4, 5] as in the input leads to the following scenario:



The first cut is done to a rod of length 7 so the cost is 7. The second cut is done to a rod of length 6 (i.e. the second part of the first cut), the third is done to a rod of length 4 and the last cut is to a rod of length 3. The total cost is 7 + 6 + 4 + 3 = 20.

Rearranging the cuts to be [3, 5, 1, 4] for example will lead to a scenario with total cost = 16 (as shown in the example photo 7 + 4 + 3 + 2 = 16).

Example 2:

Input: n = 9, cuts = [5,6,1,4,2]

Output: 22

Explanation: If you try the given cuts ordering the cost will be 25.

There are much ordering with total cost <= 25, for example, the order [4, 6, 5, 2, 1] has total cost = 22 which is the minimum possible.

Constraints:

2 <= n <= 106

1 <= cuts.length <= min(n - 1, 100)

1 <= cuts[i] <= n - 1

All the integers in cuts array are distinct.

**Solutions :**

class Solution {

    public int memo(int left,int right,int[] cuts,int[][] dp)

    {

        if(left>right) return 0;

        if(dp[left][right]!=-1) return dp[left][right];

        int cost=Integer.MAX\_VALUE;

        for(int i=left;i<=right;i++)

        {

            int c=cuts[right+1]-cuts[left-1]+memo(left,i-1,cuts,dp)+memo(i+1,right,cuts,dp);

            cost=Math.min(cost,c);

        }

        return dp[left][right]=cost;

    }

    public int tab(int[] cuts,int[][] dp)

    {

        for(int left=cuts.length-2;left>=1;left--)

        {

            for(int right=left;right<=cuts.length-2;right++)

            {

                int cost=Integer.MAX\_VALUE;

                for(int k=left;k<=right;k++)

                {

                    int c=cuts[right+1]-cuts[left-1]+dp[left][k-1]+dp[k+1][right];

                    cost=Math.min(cost,c);

                }

                dp[left][right]=cost;

            }

        }

        return dp[1][cuts.length-2];

    }

    public int minCost(int n, int[] cuts) {

        //adding 0 at start and n at end in array

        int m = cuts.length;

        Arrays.sort(cuts);

        int[] newCuts = new int[m + 2];

        System.arraycopy(cuts, 0, newCuts, 1, m);

        newCuts[m + 1] = n;

        int[][] dp = new int[m+2][m+2];

        // for(int[] i:dp)

        // {

        //     Arrayss.fill(i,-1);

        // }

        // return memo(1,m,newCuts,dp);

        return tab(newCuts,dp);

    }

}

**50) Burst Balloons(Leetcode - 312)**

You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.

If you burst the ith balloon, you will get nums[i - 1] \* nums[i] \* nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.

Return the maximum coins you can collect by bursting the balloons wisely.

Example 1:

Input: nums = [3,1,5,8]

Output: 167

Explanation:

nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []

coins = 3\*1\*5 + 3\*5\*8 + 1\*3\*8 + 1\*8\*1 = 167

Example 2:

Input: nums = [1,5]

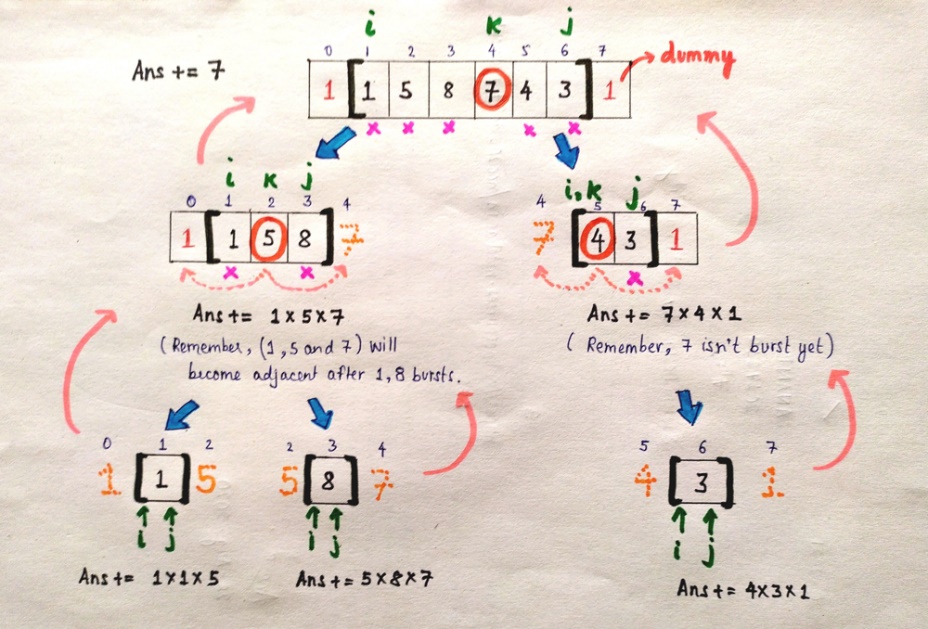
Output: 10

Constraints:

n == nums.length

1 <= n <= 300

0 <= nums[i] <= 100



**Solution :**

class Solution {

    public int memo(int left,int right,int[] nums,int[][] dp)

    {

        if(left>right) return 0;

        if(dp[left][right]!=-1) return dp[left][right];

        int max=0;

        for(int k=left;k<=right;k++)

        {

            int coins=nums[left-1]\*nums[k]\*nums[right+1]+memo(left,k-1,nums,dp)+memo(k+1,right,nums,dp);

            max=Math.max(coins,max);

        }

        return dp[left][right]=max;

    }

    public int tab(int[] nums,int[][] dp)

    {

        for(int left=nums.length-2;left>=1;left--)

        {

            for(int right=left;right<=nums.length-2;right++)

            {

                int max=0;

               for(int k=left;k<=right;k++)

               {

                    int coins=nums[left-1]\*nums[k]\*nums[right+1]+dp[left][k-1]+dp[k+1][right];

                    max=Math.max(coins,max);

                }

                dp[left][right]=max;

            }

        }

        return dp[1][nums.length-2];

    }

    public int maxCoins(int[] nums) {

        int newNums[] = new int[nums.length+2];

        newNums[0]=1;

        System.arraycopy(nums, 0, newNums, 1, nums.length);

        newNums[nums.length+1]=1;

        int[][] dp = new int[nums.length+2][nums.length+2];

        // for(int i[]:dp)

        // {

        //     Arrays.fill(i,-1);

        // }

        // return memo(1,nums.length,newNums,dp);

        return tab(newNums,dp);

    }

}

**51) Evaluate Boolean Expression to True (Striver dp – 52)**

Example 1:

Input: expression = “T|T&F”

Output: 1

Explanation: The only way to get the result as true is:

(T) | (T&F) = T|F = T

Example 2:

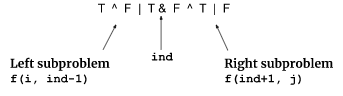
Input: expression = “F|T^F”

Output: 2

Explanation: There are 2 possible ways to get the result as true:

i) (F|T) ^ F = T ^ F = T

ii) F | (T^F) = F | T = T



**Solution : (Memoization )**

static final int MOD = 1000000007;

static long evaluateExpressionWays(String exp, int i, int j, int isTrue, Long[][][] dp) {

*// Base case 1: When the start index is greater than the end index, no ways to evaluate.*

if (i > j) {

return 0;

}

*// Base case 2: When the start and end indices are the same.*

if (i == j) {

if (isTrue == 1) {

return exp.charAt(i) == 'T' ? 1 : 0;

} else {

return exp.charAt(i) == 'F' ? 1 : 0;

}

}

if (dp[i][j][isTrue] != null) {

return dp[i][j][isTrue];

}

long ways = 0;

for (int ind = i + 1; ind <= j - 1; ind += 2) {

long lT = evaluateExpressionWays(exp, i, ind - 1, 1, dp);

long lF = evaluateExpressionWays(exp, i, ind - 1, 0, dp);

long rT = evaluateExpressionWays(exp, ind + 1, j, 1, dp);

long rF = evaluateExpressionWays(exp, ind + 1, j, 0, dp);

char operator = exp.charAt(ind);

if (operator == '&') {

if (isTrue == 1) {

ways = (ways + (lT \* rT) % MOD) % MOD;

} else {

ways = (ways + (lF \* rT) % MOD + (lT \* rF) % MOD + (lF \* rF) % MOD) % MOD;

}

} else if (operator == '|') {

if (isTrue == 1) {

ways = (ways + (lF \* rT) % MOD + (lT \* rF) % MOD + (lT \* rT) % MOD) % MOD;

} else {

ways = (ways + (lF \* rF) % MOD) % MOD;

}

} else {

if (isTrue == 1) {

ways = (ways + (lF \* rT) % MOD + (lT \* rF) % MOD) % MOD;

} else {

ways = (ways + (lF \* rF) % MOD + (lT \* rT) % MOD) % MOD;

}

}

}

dp[i][j][isTrue] = ways;

return ways;

}

static int evaluateExpWays(String exp) {

int n = exp.length();

Long[][][] dp = new Long[n][n][2]; *// dp[i][j][k] stores the number of ways to evaluate the subexpression from index i to j with the result k (0 or 1).*

return (int) evaluateExpressionWays(exp, 0, n - 1, 1, dp);

}

**Tabulation :**

static final int MOD = 1000000007;

static int evaluateExp(String exp) {

int n = exp.length();

long[][][] dp = new long[n][n][2];

*// Initializing the dp array*

for (int i = n - 1; i >= 0; i--) {

for (int j = 0; j <= n - 1; j++) {

if (i > j) continue;

for (int isTrue = 0; isTrue <= 1; isTrue++) {

*// Base case 1:*

if (i == j) {

if (isTrue == 1) dp[i][j][isTrue] = exp.charAt(i) == 'T' ? 1 : 0;

else dp[i][j][isTrue] = exp.charAt(i) == 'F' ? 1 : 0;

continue;

}

*// Recurrence logic:*

long ways = 0;

for (int ind = i + 1; ind <= j - 1; ind += 2) {

long lT = dp[i][ind - 1][1];

long lF = dp[i][ind - 1][0];

long rT = dp[ind + 1][j][1];

long rF = dp[ind + 1][j][0];

char operator = exp.charAt(ind);

if (operator == '&') {

if (isTrue == 1) ways = (ways + (lT \* rT) % MOD) % MOD;

else ways = (ways + (lF \* rT) % MOD + (lT \* rF) % MOD + (lF \* rF) % MOD) % MOD;

} else if (operator == '|') {

if (isTrue == 1) ways = (ways + (lF \* rT) % MOD + (lT \* rF) % MOD + (lT \* rT) % MOD) % MOD;

else ways = (ways + (lF \* rF) % MOD) % MOD;

} else {

if (isTrue == 1) ways = (ways + (lF \* rT) % MOD + (lT \* rF) % MOD) % MOD;

else ways = (ways + (lF \* rF) % MOD + (lT \* rT) % MOD) % MOD;

}

}

dp[i][j][isTrue] = ways;

}

}

}

return (int) dp[0][n - 1][1];

}

**52) Palindrome Partitioning II (Leetcode - 132)**

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

Example 1:

Input: s = "aab"

Output: 1

Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.

Example 2:

Input: s = "a"

Output: 0

Example 3:

Input: s = "ab"

Output: 1

Constraints:

1 <= s.length <= 2000

s consists of lowercase English letters only.

**Solution :**

class Solution {

    public int partition(int i,String s,int[] dp)

    {

        if(i==s.length())

        {

            return 0;

        }

        if(dp[i]!=-1) return dp[i];

        int min=Integer.MAX\_VALUE;

        for(int k=i;k<s.length();k++)

        {

            if(isPalindrome(i,k,s))

            {

                int cut=partition(k+1,s,dp);

                min=Math.min(min,cut);

            }

        }

        return dp[i]=1+min;  //1 for the cut

    }

    public int tab(String s)

    {

        int dp[] = new int[s.length()+1];

        dp[s.length()]=0;

        for(int i=s.length()-1;i>=0;i--)

        {

            int min=Integer.MAX\_VALUE;

            for(int k=i;k<s.length();k++)

            {

                if(isPalindrome(i,k,s))

                {

                    int cut=dp[k+1];

                    min=Math.min(min,cut);

                }

            }

            dp[i]=1+min;

        }

        return dp[0];

    }

    public boolean isPalindrome(int i, int j, String s) {

        while (i < j) {

            if (s.charAt(i) != s.charAt(j)) return false;

            i++;

            j--;

        }

        return true;

    }

    public int minCut(String s) {

        // int dp[] = new int[s.length()];

        // Arrays.fill(dp,-1);

        // return partition(0,s,dp)-1;

        return tab(s)-1;

    }

}

**53) Partition Array for Maximum Sum (Leetcode - 1043)**

Given an integer array arr, partition the array into (contiguous) subarrays of length at most k. After partitioning, each subarray has their values changed to become the maximum value of that subarray.

Return the largest sum of the given array after partitioning. Test cases are generated so that the answer fits in a 32-bit integer.

Example 1:

Input: arr = [1,15,7,9,2,5,10], k = 3

Output: 84

Explanation: arr becomes [15,15,15,9,10,10,10]

Example 2:

Input: arr = [1,4,1,5,7,3,6,1,9,9,3], k = 4

Output: 83

Example 3:

Input: arr = [1], k = 1

Output: 1

Constraints:

1 <= arr.length <= 500

0 <= arr[i] <= 109

1 <= k <= arr.length

**Solution :**

class Solution {

    public int memo(int i,int j,int[] arr,int[] dp)

    {

        if(i==arr.length)

        {

            return 0;

        }

        if(dp[i]!=-1) return dp[i];

        int maxSum=0;

        int max=0;

        int len=0;

        //subarray length should be k

        for(int k=i;k<Math.min(i+j,arr.length);k++)

        {

            max=Math.max(arr[k],max);

            len++;

            int sum = len\*max+memo(k+1,j,arr,dp);

            maxSum=Math.max(maxSum,sum);

        }

        return dp[i]=maxSum;

    }

    public int tab(int[] arr,int j)

    {

        int dp[] = new int[arr.length+1];

        dp[arr.length]=0;

        for(int i=arr.length-1;i>=0;i--)

        {

            int maxSum=0;

            int max=0;

            int len=0;

            //subarray length should be k

            for(int k=i;k<Math.min(i+j,arr.length);k++)

            {

                max=Math.max(arr[k],max);

                len++;

                int sum = len\*max+dp[k+1];

                maxSum=Math.max(maxSum,sum);

            }

            dp[i]=maxSum;

        }

        return dp[0];

    }

    public int maxSumAfterPartitioning(int[] arr, int k) {

        // int dp[] = new int[arr.length];

        // Arrays.fill(dp,-1);

        // return memo(0,k,arr,dp);

        return tab(arr,k);

    }

}

**54) Maximal Rectangle(Leetcode - 85)**

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

Example 1:

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

Output: 6

Explanation: The maximal rectangle is shown in the above picture.

Example 2:

Input: matrix = [["0"]]

Output: 0

Example 3:

Input: matrix = [["1"]]

Output: 1

Constraints:

rows == matrix.length

cols == matrix[i].length

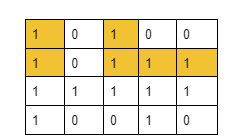
1 <= row, cols <= 200

matrix[i][j] is '0' or '1'.

**Steps: using (Area of largest rectangle in histogram)**

 we can convert each row of the given matrix into a histogram. Let’s try it out with the first row: We can consider the first and third columns to be rectangles with height 1 and the rest to be rectangles with height 0. The histogram will look like the following:

[2, 0, 2, 1, 1]



**Solution :**

class Solution {

    public int[] NSE(int[] height)

    {

        Stack<Integer> stack = new Stack<Integer>();

        int NSE[] = new int[height.length];

        for(int i=height.length-1;i>=0;i--)

        {

            while(!stack.isEmpty() && height[stack.peek()]>=height[i])

            {

                stack.pop();

            }

            NSE[i]=(!stack.isEmpty())?stack.peek():height.length;

            stack.push(i);

        }

        return NSE;

    }

    public int[] PSE(int[] height)

    {

        Stack<Integer> stack = new Stack();

        int[] PSE =new int[height.length];

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

        {

            while(!stack.isEmpty() && height[stack.peek()]>=height[i])

            {

                stack.pop();

            }

            PSE[i]=(!stack.isEmpty())?stack.peek():-1;

            stack.push(i);

        }

        return PSE;

    }

    public int findArea(int[] height)

    {

        int[] NSE = NSE(height);

        int[] PSE = PSE(height);

        int max=0;

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

        {

            int area=(NSE[i]-PSE[i]-1)\*height[i];

            max=Math.max(area,max);

        }

        return max;

    }

    public int maximalRectangle(char[][] matrix) {

       int max=0;

       int height[] = new int[matrix[0].length];

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

       {

        for(int j=0;j<matrix[0].length;j++)

        {

            if(matrix[i][j]=='1')  height[j]+=1;

            else height[j]=0;

        }

        System.out.println(Arrays.toString(height));

        max=Math.max(max,findArea(height));

       }

        return max;

    }

}

**55) Maximal Square(Leetcode - 221)**

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

Example 1:

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

Output: 4

Example 2:

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

Output: 1

Example 3:

Input: matrix = [["0"]]

Output: 0

Constraints:

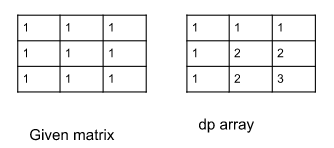
m == matrix.length

n == matrix[i].length

1 <= m, n <= 300

matrix[i][j] is '0' or '1'.

**Count No of squres : (same algo)**



Count = Sum of dp array.

Area = Max in dp array.

**Solution :**

class Solution {

    public int maximalSquare(char[][] matrix) {

        int dp[][] = new int[matrix.length+1][matrix[0].length+1];

        int maxLen=0;

        for(int i=1;i<matrix.length+1;i++)

        {

            for(int j=1;j<matrix[0].length+1;j++)

            {

                if(matrix[i-1][j-1]=='1')

                {

                    dp[i][j]=1+Math.min(Math.min(dp[i][j-1],dp[i-1][j]),dp[i-1][j-1]);

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

                }

            }

        }

        return maxLen\*maxLen;

    }

}

**56) Maximum Number of Points with Cost (Leetcode - 1937)**

You are given an m x n integer matrix points (0-indexed). Starting with 0 points, you want to maximize the number of points you can get from the matrix.

To gain points, you must pick one cell in each row. Picking the cell at coordinates (r, c) will add points[r][c] to your score.

However, you will lose points if you pick a cell too far from the cell that you picked in the previous row. For every two adjacent rows r and r + 1 (where 0 <= r < m - 1), picking cells at coordinates (r, c1) and (r + 1, c2) will subtract abs(c1 - c2) from your score.

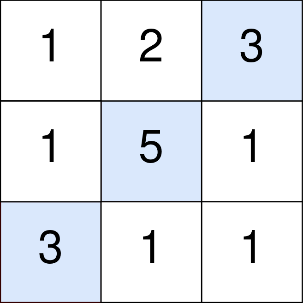
Return the maximum number of points you can achieve.

abs(x) is defined as:

x for x >= 0.

-x for x < 0.

Example 1:



Input: points = [[1,2,3],[1,5,1],[3,1,1]]

Output: 9

Explanation:

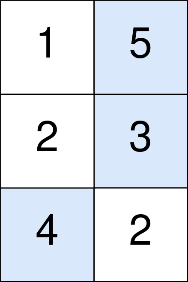
The blue cells denote the optimal cells to pick, which have coordinates (0, 2), (1, 1), and (2, 0).

You add 3 + 5 + 3 = 11 to your score.

However, you must subtract abs(2 - 1) + abs(1 - 0) = 2 from your score.

Your final score is 11 - 2 = 9.

Example 2:



Input: points = [[1,5],[2,3],[4,2]]

Output: 11

Explanation:

The blue cells denote the optimal cells to pick, which have coordinates (0, 1), (1, 1), and (2, 0).

You add 5 + 3 + 4 = 12 to your score.

However, you must subtract abs(1 - 1) + abs(1 - 0) = 1 from your score.

Your final score is 12 - 1 = 11.

Constraints:

m == points.length

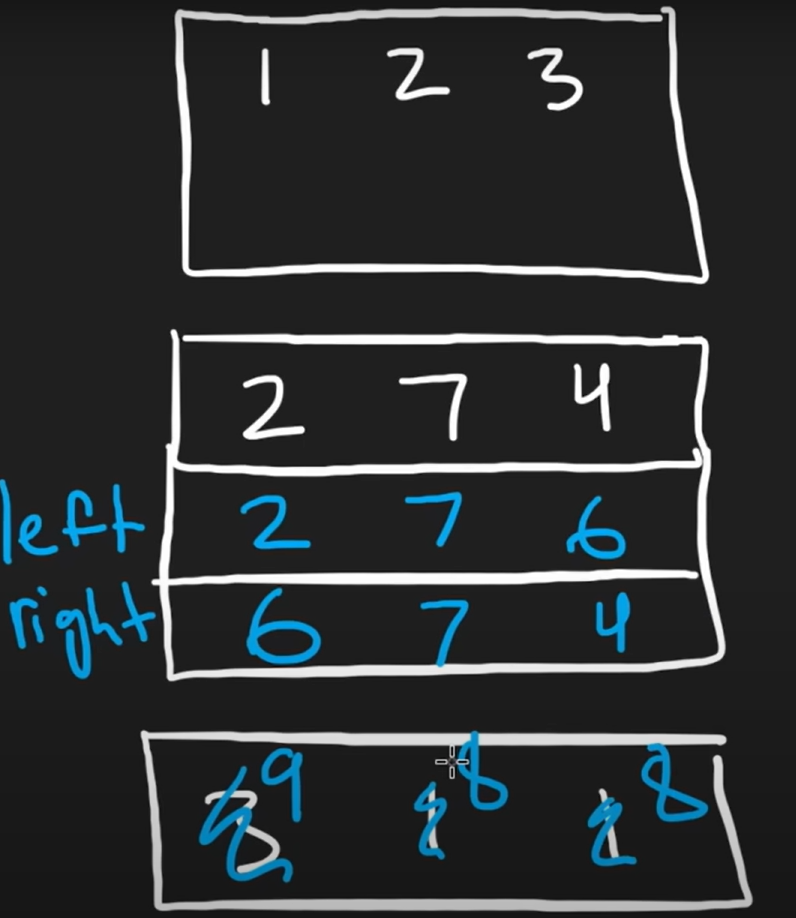
n == points[r].length

1 <= m, n <= 105

1 <= m \* n <= 105

0 <= points[r][c] <= 105

**Solution : (Tab )**

****

class Solution {

    public long memo(int i,int j,int[][] points,long[][] dp)

    {

        if(i<0) return 0;

        if(dp[i][j]!=-1) return dp[i][j];

        long max=0;

        for(int k=0;k<points[0].length;k++)

        {

            long p =  points[i][k]+memo(i-1,k,points,dp)-Math.abs(j-k);

            max=Math.max(max,p);

        }

        return dp[i][j] = max;

    }

**//different from recursion Better than recursion dp**

public long tab(int[][] points) {

    int n = points.length;

    int m = points[0].length;

    long[][] dp = new long[n][m];

    for (int j = 0; j < m; j++) {

        dp[0][j] = points[0][j];

    }

    for (int i = 1; i < n; i++) {

        long[] leftMax = new long[m];

        long[] rightMax = new long[m];

        leftMax[0] = dp[i-1][0];

        for (int j = 1; j < m; j++) {

            leftMax[j] = Math.max(leftMax[j-1]-1, dp[i-1][j]);

        }

        rightMax[m-1] = dp[i-1][m-1];

        for (int j = m-2; j >= 0; j--) {

            rightMax[j] = Math.max(rightMax[j+1]-1, dp[i-1][j]);

        }

        for (int j = 0; j < m; j++) {

            dp[i][j] = points[i][j] + Math.max(leftMax[j], rightMax[j]);

        }

    }

    long max = Long.MIN\_VALUE;

    for (int j = 0; j < m; j++) {

        max = Math.max(max, dp[n-1][j]);

    }

    return max;

}

    public long maxPoints(int[][] points) {

    //    long max=0;

    //    int n=points.length-1;

    //    long dp[][] = new long[points.length][points[0].length];

    //    for(long[] i : dp)

    //    {

    //     Arrays.fill(i,-1);

    //    }

    //    for(int j=0;j<points[0].length;j++)

    //    {

    //         long p = points[n][j]+memo(n-1,j,points,dp);

    //         max=Math.max(p,max);

    //    }

    //    return max;

    return tab(points);

    }

}

**57) Maximum Product Subarray (Leetcode - 152)**

Given an integer array nums, find a subarray that has the largest product, and return the product.

The test cases are generated so that the answer will fit in a 32-bit integer.

Example 1:

Input: nums = [2,3,-2,4]

Output: 6

Explanation: [2,3] has the largest product 6.

Example 2:

Input: nums = [-2,0,-1]

Output: 0

Explanation: The result cannot be 2, because [-2,-1] is not a subarray.

Constraints:

1 <= nums.length <= 2 \* 104

-10 <= nums[i] <= 10

The product of any subarray of nums is guaranteed to fit in a 32-bit integer.

**Solution : (using kadane)**

class Solution {

    public int maxProduct(int[] nums) {

        int n = nums.length;

        int prefix=1,suffix=1;

        int ans=nums[0];

        for(int i=0;i<n;i++)

{

            prefix \*= nums[i];   //prefix product

            suffix \*= nums[n-1-i];    //suffix product

            ans = Math.max(ans,Math.max(prefix,suffix));

            if(suffix==0) suffix=1;

            if(prefix==0) prefix=1;

        }

        return ans;

    }

}

**58) Regular Expression Matching (Leetcode - 10)**

Given an input string s and a pattern p, implement regular expression matching with support for '.' and '\*' where:

'.' Matches any single character.​​​​

'\*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Example 1:

Input: s = "aa", p = "a"

Output: false

Explanation: "a" does not match the entire string "aa".

Example 2:

Input: s = "aa", p = "a\*"

Output: true

Explanation: '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

Example 3:

Input: s = "ab", p = ".\*"

Output: true

Explanation: ".\*" means "zero or more (\*) of any character (.)".

Constraints:

1 <= s.length <= 20

1 <= p.length <= 20

s contains only lowercase English letters.

p contains only lowercase English letters, '.', and '\*'.

It is guaranteed for each appearance of the character '\*', there will be a previous valid character to match.

**Solution : (wildcard matching)**

import java.util.Arrays;

class Solution {

    // Helper method for memoization

    public boolean memo(int i, int j, String s, String p, int[][] dp) {

        // Base cases

        if (i < 0 && j < 0) return true; // Both string and pattern are exhausted

        if (j < 0) return false; // Pattern is exhausted but string is not

        if (i < 0) {

            // Handle cases where the remaining pattern could be zero or more `\*`

            while (j >= 0) {

                if (p.charAt(j) == '\*') {

                    j -= 2; // Skip the preceding character and the `\*`

                } else {

                    return false;

                }

            }

            return true;

        }

        // Check if the result is already computed

        if (dp[i][j] != -1) return dp[i][j] == 1;

        boolean match = false;

        // Check if current characters match or pattern character is `.`

        if (s.charAt(i) == p.charAt(j) || p.charAt(j) == '.') {

            match = memo(i - 1, j - 1, s, p, dp);

        }

        // Handle pattern character `\*`

        if (p.charAt(j) == '\*') {

            // Case 1: `\*` counts as zero occurrences of the preceding element skip a\* = 0 or .\* = 0

            match = match || memo(i, j - 2, s, p, dp);

            // Case 2: `\*` counts as one or more occurrences of the preceding element

            if (s.charAt(i) == p.charAt(j - 1) || p.charAt(j - 1) == '.') {

                match = match || memo(i - 1, j, s, p, dp);

            }

        }

        // Store the result in the DP table (1 for true, 0 for false)

        dp[i][j] = match ? 1 : 0;

        return match;

    }

    public boolean tab(String s,String p)

    {

        boolean[][] dp = new boolean[s.length()+1][p.length()+1];

        dp[0][0]=true;

        //s get exhausted

        int k = 2;

        while(k<=p.length())

        {

            if(p.charAt(k-1)=='\*')

            {

                dp[0][k]=true;

                k+=2;

            }

            else

            {

                break;

            }

        }

        for(int i=1;i<=s.length();i++)

        {

            for(int j=1;j<=p.length();j++)

            {

                boolean match=false;

                 if (s.charAt(i-1) == p.charAt(j-1) || p.charAt(j-1) == '.')

                 {

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

                 }

                if (p.charAt(j-1) == '\*')

                {

                    match = match || dp[i][j - 2];

                    if (s.charAt(i-1) == p.charAt(j - 2) || p.charAt(j - 2) == '.')

                    {

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

                    }

                }

                dp[i][j] = match;

            }

        }

        return dp[s.length()][p.length()];

    }

    public boolean isMatch(String s, String p) {

        // Initialize DP table with -1 (indicating uncomputed state)

        int[][] dp = new int[s.length()][p.length()];

        for (int[] row : dp) {

            Arrays.fill(row, -1);

        }

        // Start recursion from the last indices of s and p

        return memo(s.length() - 1, p.length() - 1, s, p, dp);

        // return tab(s,p);

    }

}