**Q1. Stairs**

You are climbing a staircase and it takes **A** 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?

Return the number of distinct ways modulo 1000000007

**Logic -**

This is the most basic dynamic programming problem.  
We know that we can take 1 or 2 step at a time. So, to take n steps,  
we must have arrived at it immediately from (n-1) or (n-2) step.  
If we knew the number of ways to reach (n-1) and (n-2) step,  
our answer would be the summation of their number of ways.

**Code -**

public class Solution {

public long helper(int A, long[] dp, long mod){

if(A<=2) return A;

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

dp[A]= (helper(A-1, dp, mod) % mod + helper(A-2, dp, mod) % mod) % mod;

return dp[A];

}

public int climbStairs(int A) {

if(A<=2) return A;

long mod = (1000\*1000\*1000+7);

long[] dp = new long[A+1];

Arrays.fill(dp, -1);

return (int) (helper(A, dp, mod) % (mod));

}

}

**Q2. Minimum Number of Squares**

Given an integer **A**. Return **minimum** count of numbers, sum of whose **squares** is equal to **A**.

**Logic –** We give each number from 1

If n <= 1,   
then return n   
Else  
 countMinSquares(n) = min {1 + countMinSquares(n - i\*i)}   
 where i >= 1 and i\*i <= n

**Code -**

public class Solution {

public int helper(int A, int[] dp) {

if(A==0) return 0;

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

int ans = Integer.MAX\_VALUE;

int i=1;

while(i\*i<=A) {

int x = helper(A-(i\*i), dp);

ans = Math.min(ans, x);

i+=1;

}

return dp[A] = ans+1;

}

public int countMinSquares(int A) {

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

Arrays.fill(dp, -1);

return helper(A, dp);

}

}

**Q3. Max Sum Without Adjacent Elements**

Given a **2 x N** grid of integer, **A**, choose numbers such that the **sum** of the numbers is maximum and no two chosen numbers are adjacent horizontally, vertically or diagonally, and return it.

**Note:** You can choose more than 2 numbers.

**Logic -**

1. Create an array with max values of the column.
2. Recursive relation – dp[i] = max(dp[i+1], dp[i+2] + A[i])
3. Basically we have 2 options one is to select the ith element or reject it.
4. If we reject it we have the next complete array to check.
5. Else if it is accepted we cant select the next element ie (i+1)th so we select the next part of array starting from (i+2)th element.

**Code -**

public class Solution {

public int helper(int[] A, int i, int[]dp) {

if(i>=A.length) return 0;

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

int x = helper(A, i+1, dp);

int y = A[i] + helper(A, i+2, dp);

return dp[i] = Math.max(x, y);

}

public int adjacent(int[][] A) {

int[] B = new int[A[0].length];

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

B[i] = Math.max(A[0][i], A[1][i]);

}

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

Arrays.fill(dp, -1);

return helper(B, 0, dp);

}

}

**Q4. Maximum Sum Value**

You are given an array **A** of **N** integers and three integers **B, C, and D**.

You have to find the maximum value of **A[i]\*B + A[j]\*C + A[k]\*D**, where **1 <= i <= j <= k <= N**.

**Logic – Maximize each term at a time.**

1. We maintain a pref array in which we add the expression A[i] \* B. Then take a prefix sum for the array. Now, at any index i this array will give the max value of A[i] \* B.
2. Then we add the value of A[j] \* C in array and again take prefix sum of the array. Now, at any index i this array will give the max value of A[i] \* B + A[j]\*C.
3. Similarly, we repeat for the 3 terms and return the last element of the array.

**Code -**

public class Solution {

public int solve(ArrayList<Integer> A, int B, int C, int D) {

int[] pref = new int[A.size()];

int[] arr = {B, C, D};

Arrays.fill(pref, 0);

for(int x:arr) {

for(int i=0;i<A.size();i++) pref[i] += A.get(i) \* x;

for(int i=1;i<A.size();i++) pref[i] = Math.max(pref[i], pref[i-1]);

}

return pref[A.size()-1];

}

}

**Q5. Max Product Subarray**  
  
Given an integer array **A** of size **N**. Find the contiguous subarray within the given array (containing at least one number) which has the largest product.

Return an integer corresponding to the **maximum** product possible.

**NOTE:** Answer will fit in 32-bit integer value.

**Logic –** Maintain a min variable. If a negative number is encountered that min variable can become the maximum. So, at every iteration we maintain max and min of A[i], mx\*A[i], mn\*A[i].

If there were no zeros or negative numbers, then the answer would definitely be the product of the whole array.

Now lets assume there were no negative numbers and just positive numbers and 0. In that case we could maintain a current maximum product which would be reset to A[i] when 0s were encountered.  
When the negative numbers are introduced, the situation changes ever so slightly. We need to now maintain the maximum product in positive and maximum product in negative. On encountering a negative number, the maximum product in negative can quickly come into picture.

**Code -**

public class Solution {

public int maxProduct(final List<Integer> A) {

int mx = A.get(0);

int mn = A.get(0);

int ans = A.get(0);

for(int i=1;i<A.size();i++) {

int temp = mn;

mn = Math.min(A.get(i), Math.min(mx\*A.get(i), mn\*A.get(i)));

mx = Math.max(A.get(i), Math.max(mx\*A.get(i), temp\*A.get(i)));

ans = Math.max(ans, mx);

}

return ans;

}

}

**Q6. Ways to Decode**

A message containing letters from **A-Z** is being encoded to numbers using the following mapping:

'A' -> 1  
'B' -> 2  
...  
'Z' -> 26

Given an encoded message denoted by string **A** containing digits, determine the total number of ways to decode it modulo **109 + 7**.

**Logic –** Consider all single digit then consider all 2 digits only if they are between 10 - 26

if (isValid(s[startIndex])) answer += ways(s, startIndex + 1);  
 if (isValid(s[startIndex] + s[startIndex + 1])) answer += ways(s, startIndex + 2);  
 return answer;  
 }

**Code -**

public class Solution {

public long helper(String A, int i, HashMap<String, Long> map) {

if(i == A.length()) return 1;

if(A.charAt(i) == '0') return 0;

if(map.containsKey(A.substring(i, A.length()))) return map.get(A.substring(i, A.length()));

HashSet<Character> temp = new HashSet<Character>();

temp.add('0');

temp.add('1');

temp.add('2');

temp.add('3');

temp.add('4');

temp.add('5');

temp.add('6');

long mod = (1000 \* 1000 \* 1000 + 7);

// Single digit

long x = helper(A, i+1, map);

// Check for 2 digit numbers from 10 to 26

if(i+1 < A.length() && (A.charAt(i) == '1' || (A.charAt(i) == '2' && temp.contains(A.charAt(i+1)))))

x += helper(A, i+2, map);

map.put(A.substring(i, A.length()), (x) % mod);

return (x)%mod;

}

public int numDecodings(String A) {

HashMap<String, Long> map = new HashMap<String, Long>();

return (int)helper(A, 0, map);

}

}

**Q 7 Unique Paths in a Grid**

Given a grid of size **n \* m**, lets assume you are starting at **(1,1)** and your goal is to reach **(n, m)**. At any instance, if you are on **(x, y)**, you can either go to **(x, y + 1)** or **(x + 1, y)**.

Now consider if some obstacles are added to the grids. How many unique paths would there be? An obstacle and empty space is marked as **1** and **0** respectively in the grid.

**Logic -**

dp[i][j] = dp[i-1][j] + dp[i][j-1] if (i-1, j) and (i, j-1) doesn’t have obstacles.  
= dp[i-1][j] if only (i-1, j) doesn’t have obstacle  
= dp[i][j-1] if only (i, j-1) doesn’t have obstacle  
= 0 otherwise

**Code -**

public class Solution {

public int helper(int[][] A, int i, int j, int[][] dp) {

if(i>=A.length || j>=A[0].length) return 0;

if(A[i][j] == 1) return 0;

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

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

return dp[i][j] = helper(A, i+1, j, dp) + helper(A, i, j+1, dp);

}

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

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

for(int i=0;i<A.length;i++) Arrays.fill(dp[i], -1);

return helper(A, 0, 0, dp);

}

}

**Q8. N digit numbers**

Find out the number of **A** digit positive numbers, whose digits on being added equals to a given number **B**.

Note that a valid number starts from digits 1-9 except the number 0 itself. i.e. leading zeroes are not allowed.

Since the answer can be large, output answer modulo 1000000007

**Logic -**

Dp[A][B] = dp[A-1][B-i], where I goes from 0 to 9

Only care to be taken is that the first digit of any number cannot be 0. So, we have used a boolean value to denote the first iteration.

**Code -**

public class Solution {

public int helper(int A, int B, boolean flag, int[][] dp) {

if(B > (9\*A)) return 0;

if(A==0 && B==0) return 1;

if(A==0 && B!=0) return 0;

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

long mod = (1000 \* 1000 \* 1000 + 7);

long x = 0;

for(int i=0;i<10;i++) {

if(B<i) break;

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

x += helper(A-1, B-i, false, dp);

x%=mod;

}

dp[A][B] = (int)(x%mod);

return (int)(x%mod);

}

public int solve(int A, int B) {

int dp[][] = new int[1001][10001];

for (int i = 0; i < A + 1; i++) {

for (int j = 0; j < B + 1; j++)

dp[i][j] = -1;

}

return helper(A, B, true, dp);

}

}

**Q9. Min Sum Path in Triangle**

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

Adjacent numbers for jth number of row i is jth and (j+1)th numbers of row i+1 is

**Logic -**

minpath[k][i] = min( minpath[k+1][i], minpath[k+1][i+1]) + triangle[k][i];

**Code -**

public class Solution {

public int helper(ArrayList<ArrayList<Integer>> a, int i, int j, int[][] dp) {

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

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

int x = a.get(i).get(j);

int y = a.get(i).get(j);

x += helper(a, i+1, j, dp);

y += helper(a, i+1, j+1, dp);

return dp[i][j] = Math.min(x, y);

}

public int minimumTotal(ArrayList<ArrayList<Integer>> a) {

int [][] dp = new int[a.size()][a.get(a.size()-1).size()];

for(int i=0;i<a.size();i++) Arrays.fill(dp[i], -1);

return helper(a, 0, 0, dp);

}

}

**Q10. Dungeon Princess**

The demons had captured the **princess** and imprisoned her in the **bottom-right** corner of a dungeon. The dungeon consists of **M x N** rooms laid out in a 2D grid. Our valiant **knight** was initially positioned in the **top-left** room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only **rightward or downward** in each step.

Given a 2D array of integers **A** of size **M x N**. Find and return the knight's **minimum** initial health so that he is able to rescue the princess.

**Logic -**

hp[i][j] = max(1, min(hp[i][j+1], hp[i+1][j]) - A[i][j])

**Code -**

public class Solution {

public int helper(ArrayList<ArrayList<Integer>> a, int i, int j, int[][] dp) {

int n = a.size();

int m = a.get(0).size();

if(i==n-1 && j==m-1) return Math.max(1, 1-a.get(i).get(j)); // For the last block

if(i==n || j==m) return Integer.MAX\_VALUE; // For the edge blocks

if(dp[i][j] != -1) return dp[i][j]; // If already solved

int right = helper(a, i, j+1, dp);

int down = helper(a, i+1, j, dp);

return dp[i][j] = Math.max(1, Math.min(right, down) - a.get(i).get(j));

}

public int calculateMinimumHP(ArrayList<ArrayList<Integer>> a) {

int n = a.size();

int m = a.get(0).size();

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

for(int i=0;i<n;i++) Arrays.fill(dp[i], -1);

return helper(a, 0, 0, dp);

}

}

**Q11. Min Sum Path in Matrix**  
  
Given a M x N grid **A** of integers, find a path from top left to bottom right which **minimizes** the sum of all numbers along its path.

Return the **minimum** sum of the path.

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

**Logic -**

DP[i][j] = A[i][j] + min(DP[i-1][j],DP[i][j-1]).

**Code -**

public class Solution {

public int helper(ArrayList<ArrayList<Integer>> A, int i, int j, int[][] dp) {

int n = A.size();

int m = A.get(0).size();

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

if(i>=n || j>=m) return Integer.MAX\_VALUE;

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

int right = helper(A, i, j+1, dp);

int down = helper(A, i+1, j, dp);

return dp[i][j] = A.get(i).get(j) + Math.min(right, down);

}

public int minPathSum(ArrayList<ArrayList<Integer>> A) {

int n = A.size();

int m = A.get(0).size();

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

for(int i=0;i<n;i++) Arrays.fill(dp[i], -1);

return helper(A, 0, 0, dp);

}

}

**Q12. Let's Party**

In Danceland, one person can party either alone or can pair up with another person.

Can you find in how many ways they can party if there are **A** people in Danceland?

**Note:** Return your answer modulo 10003, as the answer can be large.

**Logic -**

Number\_of\_ways(k-1) + (k-1) \* Number\_of\_ways(k-2)

**Code -**

public class Solution {

public long helper(int A, long[] dp) {

if(A==1) return 1;

if(A==2) return 2;

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

return dp[A] = (helper(A-1, dp) + helper(A-2, dp)\*(A-1)) % 10003;

}

public int solve(int A) {

long[] dp = new long[A+1];

Arrays.fill(dp, -1);

return (int) (helper(A, dp) % 10003);

}

}

**Q13. 0-1 Knapsack**

Given two integer arrays **A** and **B** of size **N** each which represent **values** and **weights** associated with **N** items respectively.

Also given an integer **C** which represents knapsack capacity.

Find out the maximum value subset of **A** such that sum of the weights of this subset is smaller than or equal to **C**.

**NOTE:** You cannot break an item, either pick the complete item, or don’t pick it

**Logic -** dp[i][j]=max(dp[i-1][j] (When we don’t consider this item) or dp[i-1][j-wt[i]]+val[i] (When we consider this item) )

**Code -**

public class Solution {

public int helper(ArrayList<Integer> A, ArrayList<Integer> B, int C, int N, int[][] dp) {

if(N==0 || C==0) return 0;

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

if(C>=B.get(N-1)) {

dp[N][C] = Math.max(A.get(N-1)+helper(A, B, C-B.get(N-1), N-1, dp), helper(A, B, C, N-1, dp));

} else {

dp[N][C] = helper(A, B, C, N-1, dp);

}

return dp[N][C];

}

public int solve(ArrayList<Integer> A, ArrayList<Integer> B, int C) {

int n = A.size();

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

for(int i=0;i<=n;i++) Arrays.fill(dp[i], -1);

int x = helper(A, B, C, A.size(), dp);

int i = B.size();

int j = C;

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

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

else {

i-=1;

if(i>=0) j-=B.get(i);

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

System.out.print(i);

System.out.print(' ');

}

}

}

return x;

}

}

**Q14. Unbounded Knapsack**

**Same as above but the relation changes a bit as we have the option to pick the item again.**

dp[i][j]=max(dp[i-1][j] (When we don’t consider this item) or dp[i][j-wt[i]]+val[i] (When we consider this item) )

**Q15. Flip Array**

Given an array **A** of positive elements, you have to flip the sign of some of its elements such that the resultant **sum** of the elements of array should be **minimum non-negative(as close to zero as possible)**.

Return the **minimum number** of elements whose sign needs to be flipped such that the resultant sum is **minimum non-negative**.

**Logic –** Find the number of elements with sum close to Sum / 2

Let the sum of all the given elements be S.

This problem can be reduced to a Knapsack problem where we have to fill a Knapsack of capacity (S/2) as fully as possible and using the minimum no. of elements. We will fill the Knapsack with the given elements.

Sign of all the elements which come into the knapsack will be flipped. As sum of all the elements in the Knapsack will be as close to S/2 as possible, we are indirectly calculating minimum non-negative sum of all the elements after flipping the sign.

**Code -**

public class Solution {

// DO NOT MODIFY THE LIST. IT IS READ ONLY

public class Node{

int sum;

int items;

public Node(int s, int i) {

sum = s;

items = i;

}

}

public Node compare (Node x, Node y) {

if(x.sum == y.sum) {

if(x.items < y.items) return x;

else return y;

}

if(x.sum > y.sum) return x;

return y;

}

public Node helper(List<Integer> A, int n, int k, Node[][] dp) {

if(n==0 || k==0) return new Node(0, 0);

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

Node x = helper(A, n-1, k, dp);

if(k>=A.get(n-1)) {

Node y = helper(A, n-1, k-A.get(n-1), dp);

Node include = new Node(A.get(n-1) + y.sum, 1+y.items);

return dp[n][k] = compare(include, x);

}

return dp[n][k] = x;

}

public int solve(final List<Integer> A) {

int k = 0;

for(int i=0;i<A.size();i++) k += A.get(i);

Node[][] dp = new Node[A.size()+1][k/2+1];

for(int i=0;i<=A.size();i++) Arrays.fill(dp[i], new Node(-1, -1));

return helper(A, A.size(), k/2, dp).items;

}

}

**Q16. Tushar's Birthday Party**

As it is Tushar's Birthday on March 1st, he decided to throw a party to all his friends at TGI Fridays in Pune. Given are the eating capacity of each friend, filling capacity of each dish and cost of each dish. A friend is satisfied if the sum of the filling capacity of dishes he ate is equal to his capacity. Find the minimum cost such that all of Tushar's friends are satisfied (reached their eating capacity).

**NOTE**:

Each dish is supposed to be eaten by only one person. Sharing is not allowed.

Each friend can take any dish unlimited number of times.

There always exists a dish with filling capacity 1 so that a solution always exists.

**Logic -**

1. Find the child with maximum eating capacity, then do the 0-1 knapsack but with tabular method as memoization doesn’t give result for all eating capacities.
2. Later on the last row of the dp column will have the costs for all eating capacities.
3. Simply add the eating capacities for each student given in A.

**Code -**

public class Solution {

public int solve(final List<Integer> A, final List<Integer> B, final List<Integer> C) {

int maxA = A.get(0);

for(int i=1;i<A.size();i++) maxA = Math.max(maxA, A.get(i));

long[][] dp = new long[B.size()+1][maxA+1];

Arrays.fill(dp[0], Integer.MAX\_VALUE);

for(int i=0;i<=B.size();i++) {

dp[i][0] = 0;

}

for(int i=1;i<=B.size();i++) {

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

if(j>=B.get(i-1)){

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

} else {

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

}

}

}

int x = 0;

for(int i=0;i<A.size();i++) {

x+=dp[B.size()][A.get(i)];

}

return x;

}

}

**Q17. Ways to send the signal**

You are trying to send signals to aliens using a linear array of **A** laser lights. You don't know much about how the aliens are going to percieve the signals, but what you know is that if two consecutive lights are on then the aliens might take it as a sign of danger and destroy the earth.

Find and return the total number of ways in which you can send a signal without compromising the safty of the earth. Return the ans % 109 + 7.

**Logic -**

Dp[A] = Dp[A-1] + Dp[A-2]

**Code -**

public class Solution {

public int helper(int A, int[] dp) {

if(A==0) return 0;

if(A==1) return 2;

if(A==2) return 3;

int mod = (1000\*1000\*1000+7);

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

return dp[A] = (helper(A-1, dp) % mod + helper(A-2, dp) % mod) % mod;

}

public int solve(int A) {

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

Arrays.fill(dp, -1);

return helper(A, dp);

}

}

**Q18. 0-1 Knapsack II**

Given two integer arrays **A** and **B** of size **N** each which represent **values** and **weights** associated with **N** items respectively.

Also given an integer **C** which represents knapsack capacity.

Find out the **maximum value** subset of **A** such that sum of the weights of this subset is smaller than or equal to **C**.

**NOTE:** You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).

**Logic -**

Since the value of items will be less than equal to 50. So the max value can be 50 \* N.

Create a dp array of size 50 \* N where dp[val] will tell that minimum weight require to have value exactly equal to val.

This can be easily calculated by running two loops:

for i -> 0 to N-1:  
 for val -> mxval to A[i]  
 dp[val] = min(dp[val],B[i] + dp[val-A[i]])

Now, check for the maximum value for which dp[val] is less than equal to the capacity of knapsack.

**Code -**

public class Solution {

public int solve(ArrayList<Integer> A, ArrayList<Integer> B, int C) {

int sumA = 0;

for(int i=0;i<A.size();i++) sumA+=A.get(i);

long[] dp =new long[sumA+1];

for(int i=0;i<=sumA;i++) dp[i] = Integer.MAX\_VALUE;

dp[0] = 0;

for(int i=1;i<=A.size();i++) {

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

if(j>=A.get(i-1)) {

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

}

}

}

int ans = 0;

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

if(dp[i] <= C && dp[i] >= 0) {

ans = i;

break;

}

}

return ans;

}

}

**Q19. Cutting a Rod**  
  
Given a rod of length **N** units and an array **A** of size N denotes prices that contains prices of all pieces of size **1 to N**.

Find and return the **maximum** value that can be obtained by cutting up the rod and selling the pieces.

**Logic -**

For the naive solution, We can get the best price by making a cut at different positions and comparing the values obtained after a cut.  
We can recursively call the same function for a piece obtained after a cut.

Let’s optimize it.

Consider val[i] denotes the maximum price we can get by selling the rod (can sell by cutting any how) of length i.  
->for all j from 0 to i-1, val[i] = max(val[i],A[j] + val[i-j-1]).

val[n] will be the output.

**Code -**

public class Solution {

public int helper(ArrayList<Integer> A, int N, int[]dp) {

if(N==0) return 0;

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

int res = 0;

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

int x = A.get(i-1) + helper(A, N-i, dp);

res = Math.max(res, x);

}

return dp[N] = res;

}

public int solve(ArrayList<Integer> A) {

int[] dp = new int[A.size()+1];

Arrays.fill(dp, -1);

return helper(A, A.size(), dp);

}

}

**Q20. Coin Sum Infinite**

You are given a set of coins **A**. In how many ways can you make sum **B** assuming you have infinite amount of each coin in the set.

**NOTE:**

* Coins in set A will be unique. Expected space complexity of this problem is **O(B)**.
* The answer can overflow. So, return the answer % (106 + 7).

**Logic –** Same as infinite Knapsack

**Code -**

public class Solution {

public int coinchange2(ArrayList<Integer> A, int B) {

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

int mod = (1000\*1000+7);

Arrays.fill(dp, 0);

dp[0] = 1;

for(int i=1;i<=A.size();i++) {

for(int j=A.get(i-1);j<=B;j++) {

dp[j] += dp[j-A.get(i-1)];

dp[j] %= mod;

}

}

return dp[B];

}

}

**Q21. Distinct Subsequences**

Given two sequences **A** and **B**, count number of unique ways in sequence **A**, to form a subsequence that is identical to the sequence **B**.

**Subsequence :** A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ACE" is a subsequence of "ABCDE" while "AEC" is not).

**Logic -**

case 1). if B[i] != A[j], then the solution would be to ignore the character A[j] and align substring B[0..i] with A[0..(j-1)]. Bherefore, dp[i][j] = dp[i][j-1].

case 2). if B[i] == A[j], then first we could adopt the solution in case 1), but also we could match the characters B[i] and A[j] and align the rest of them (i.e. B[0..(i-1)] and A[0..(j-1)]. As a result, dp[i][j] = dp[i][j-1] + d[i-1][j-1]

**Code -**

public class Solution {

public int numDistinct(String A, String B) {

int n = A.length();

int m = B.length();

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

Arrays.fill(dp, 0);

dp[0] = 1;

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

for(int j=m;j>=1;j--) {

if(A.charAt(i-1) == B.charAt(j-1)) dp[j] = dp[j] + dp[j-1];

}

}

return dp[m];

}

}

**Q22. Length of Longest Fibonacci Subsequence**

Given a strictly increasing array **A** of positive integers forming a sequence.

A sequence **X1, X2, X3, ..., XN** is fibonacci like if

* **N > =3**
* **Xi + Xi+1 = Xi+2 for all i+2 <= N**

Find and return the length of the longest Fibonacci-like subsequence of **A**.

If one does not exist, return 0.

**NOTE:** A subsequence is derived from another sequence A by deleting any number of elements (including none) from A, without changing the order of the remaining elements.

**Logic -**

**Code -**

class Solution:

# @param A : list of integers

# @return an integer

def solve(self, A):

if(len(A)<3): return 0

hashA = {}

for i in range(len(A)):

hashA[A[i]] = i

ans = 0

dp = [[0 for \_ in range(len(A))] for \_ in range(len(A))]

for k in range(len(A)):

for j in range(k):

if(A[k]-A[j] in hashA and A[k]-A[j] < A[j]):

i = hashA[A[k]-A[j]]

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

ans = max(ans, dp[j][k]+2)

return ans

**Q23. Longest Palindromic Subsequence**

Given a string **A**. Find the longest palindromic subsequence (A subsequence which does not need to be contiguous and is a palindrome).

You need to return the **length** of longest palindromic subsequence.

**Logic -**

-> If last and first character of the sequence are same, then L(0,n-1) = L(1,n-2) + 2  
->Else, L(0,n-1) = max(L(0,n-2),L(1,n-1))

**Code -**

class Solution:

def helper(self, A, n, m, dp):

if(n>m): return 0

if(n==m): return 1

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

if(A[n]==A[m]):

dp[n][m] = 2 + self.helper(A, n+1, m-1, dp)

return dp[n][m]

dp[n][m] = max(self.helper(A, n, m-1, dp), self.helper(A, n+1, m, dp))

return dp[n][m]

def solve(self, A):

dp = [[-1 for \_ in range(len(A)+1)] for \_ in range(len(A)+1)]

x = self.helper(A, 0, len(A)-1, dp)

return x

**Q24. Matrix Chain Multiplication**

Given an array of integers **A** representing chain of **2-D** matices such that the dimensions of **ith** matrix is **A[i-1] x A[i]**.

Find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Return the **minimum number** of multiplications needed to multiply the chain.

**Logic -**

Matrix Multiplication is associative, if we have four matrices W,X,Y and Z.  
WXYZ = W(XY)Z = (WX)((YZ) = (W)(X(YZ))  
The Problem reduces to place parenthesis such that the cost is minimum.

For example, if the given chain is of 4 matrices. let the chain be WXYZ,  
then there are 3 ways to place first set of parenthesis outer side: (W)(XYZ), (WX)(YZ) and (WXY)(Z).  
So when we place a set of parenthesis, we divide the problem into subproblems of smaller size.

**Helper(I, j) = Helper(I, k) + Helper(k+1, j) + A[I-1] \* A[j] \* A[k], where k = I to j**

If I==j return 0

TC = O(N^2)

**Code -**

public class Solution {

public int minCost(ArrayList<Integer> A, int i, int j, int[][] dp) {

if(i==j) return 0;

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

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

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

dp[i][j] = Math.min(dp[i][j], minCost(A, i, k, dp) + minCost(A, k+1, j, dp) + A.get(i-1)\*A.get(j)\*A.get(k));

}

return dp[i][j];

}

public int solve(ArrayList<Integer> A) {

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

for(int i=0;i<A.size();i++) Arrays.fill(dp[i], -1);

return minCost(A, 1, A.size()-1, dp);

}

}

**Q25. Russian Doll Envelopes**

Given a matrix of integers **A** of size **N x 2** describing dimensions of **N** envelopes, where **A[i][0]** denotes the height of the **ith** envelope and **A[i][1]** denotes the width of the **i**th envelope.

One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

Find the **maximum number** of envelopes you can put one inside other.

**Logic -**

1. Sort the array on the basis of width.
2. Then apply LIS on the height. Also add a condition for width, as envelopes with same width are not acceptable.

We can also sort on the basis of height

**Code -**

class Solution:

def solve(self, A):

A.sort()

n = len(A)

dp = [1 for \_ in range(n)]

ans = 1

dp[0] = 1

for i in range(1, n):

res = 0

for j in range(i):

if(A[i][0]>A[j][0] and A[i][1]>A[j][1]): res = max(res, dp[j])

dp[i] = res + 1;

ans = max(ans, res+1)

return ans

**Q26. Interleaving Strings**

Given **A, B, C** find whether **C** is formed by the interleaving of **A** and **B**.

Logic -

There are 3 posibble cases at any time if we compare A, B, C character wize.

1. If A[i] = C[i], then we match that character and move forward.
2. Same goes for B[i] = C[i]
3. If both are same then we have to check both the condition ie by taking a character from A first then taking a character from B.

Code -

class Solution:

# @param A : string

# @param B : string

# @param C : string

# @return an integer

def isInterleave(self, A, B, C):

if(len(A) + len(B) != len(C)): return 0

dp = [[0 for \_ in range(len(B)+1)] for \_ in range(len(A)+1)]

for i in range(len(A)+1):

for j in range(len(B)+1):

# 2 empty strings

if(i==0 and j==0): dp[i][j] = 1

# 1st row

elif (i==0 and B[j-1]==C[j-1]): dp[i][j] = dp[i][j-1]

# 1st col

elif (j==0 and A[i-1]==C[i-1]): dp[i][j] = dp[i-1][j]

# One of the character is equal to C[i+j-1]

elif (A[i-1]==C[i+j-1] and B[j-1]!=C[i+j-1]): dp[i][j] = dp[i-1][j]

elif (A[i-1]!=C[i+j-1] and B[j-1]==C[i+j-1]): dp[i][j] = dp[i][j-1]

# Both characters are equal

elif (A[i-1]==C[i+j-1] and B[j-1]==C[i+j-1]): dp[i][j] = (dp[i-1][j] or dp[i][j-1])

return dp[len(A)][len(B)]

**Q27. Palindrome Partitioning II**

Given a string **A**, partition **A** such that every substring of the partition is a palindrome.

Return the **minimum cuts** needed for a palindrome partitioning of A.

**Logic -**

**Code -**

class Solution:

# @param A : string

# @return an integer

def minCut(self, A):

isPal = [[-1 for \_ in range(len(A))] for \_ in range(len(A))]

for i in range(len(A)-1, -1, -1):

for j in range(i, len(A)):

if i==j:

isPal[i][j] = True

continue

elif j==i+1 and A[i]==A[j]:

isPal[i][j] = True

elif j==i+1 and A[i]!=A[j]:

isPal[i][j] = False

else:

if A[i]==A[j]: isPal[i][j] = isPal[i+1][j-1]

else: isPal[i][j] = False

dp = [0 for i in range(len(A))]

for i in range(1, len(A)):

if(isPal[0][i]):

dp[i] = 0

continue

res = 2000

for j in range(i+1):

if(isPal[j][i]):

res = min(res, 1 + dp[j-1])

dp[i] = res

return dp[len(A)-1]

**Q28. Edit Distance**  
  
Given two strings **A** and **B**, find the **minimum** number of steps required to convert **A** to **B**. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

* Insert a character
* Delete a character
* Replace a character

**Logic -**

We look at the first character of both the strings.  
If they match, we can look at the answer from remaining part of S1 and S2.  
If they don’t, we have 3 options.  
1) Insert S2’s first character and then solve the problem for remaining part of S2, and S1.  
2) Delete S1’s first character and trying to match S1’s remaining string with S2.  
3) Replace S1’s first character with S2’s first character in which case we solve the problem for remaining part of S1 and S2.

**Code -**

public class Solution {

public int minDistance(String A, String B) {

int[][] dp = new int[A.length()+1][B.length()+1];

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

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

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

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

if(A.charAt(i-1) == B.charAt(j-1)) {

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

} else {

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

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

}

}

}

return dp[A.length()][B.length()];

}

}

**Q29. Longest Common Subsequence**  
  
Given two strings **A** and **B**. Find the **longest common subsequence** ( A sequence which does not need to be contiguous), which is common in both the strings.

You need to return the **length** of such longest common subsequence.

**Logic -**

LCS(i, j) = maximum (LCS(i-1, j, LCS(i, j-1) if A[i] != B[j]

LCS(i, j) = 1 + LCS(I-1, j-1) if A[i] = B[j]

**Code -**

class Solution:

# @param A : string

# @param B : string

# @return an integer

def solve(self, A, B):

dp = [[0 for i in range(len(A)+1)] for i in range(len(B)+1)]

for i in range(1, len(A)+1):

for j in range(1, len(B)+1):

if A[i-1] == B[j-1]:

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

else:

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

return dp[len(A)][len(B)]

**Q30. Regular Expression Match**

Implement wildcard pattern matching with support for **' ? '** and **' \* '** for strings **A** and **B**.

* ' ? ' : Matches any single character.
* ' \* ' : Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

**Logic -**

There are 3 cases for this problem.

1. A[i] == B[j] or B[j] == ‘?’ -> Match both the characters and check for the rest of the string. Dp[i][j] = dp[I-1][j-1]
2. B[j] = \* -> There are 2 cases in this, one is to leave the \* and second is to match one character in A with \* and check for next characters. Dp[i][j] = dp[I-1][j] || dp[i][j-1]
3. If A[i] != B[j] return false.

**Code -**

class Solution:

# @param A : string

# @param B : string

# @return an integer

def isMatch(self, A, B):

dp = [[False for \_ in range(len(B) + 1)] for \_ in range(len(A) + 1)]

i = 0

while (i <= len(A)) :

if (i == 0) :

dp[i][0] = True

else :

dp[i][0] = False

i += 1

i = 1

while (i <= len(B)) :

if (B[i - 1] == '\*') :

dp[0][i] = dp[0][i - 1]

else :

dp[0][i] = False

i += 1

i = 1

while (i <= len(A)) :

j = 1

while (j <= len(B)) :

if (A[i - 1] == B[j - 1] or B[j - 1] == '?') :

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

elif(B[j - 1] == '\*') :

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

else :

dp[i][j] = False

j += 1

i += 1

if dp[len(A)][len(B)] == True: return 1

return 0

**Q31. Unique Binary Search Trees II**

Given an integer **A**, how many structurally unique BST's (binary search trees) exist that can store values 1...A?

**Logic – Catalyn series**

1. Lets say you know the answer for values i which ranges from 0 <= i <= n - 1.  
   How do you calculate the answer for n.
2. Lets consider the number [1, n]  
   We have n options of choosing the root.  
   If we choose the number j as the root, j - 1 numbers fall in the left subtree, n - j numbers fall in the right subtree. We already know how many ways there are to forming j - 1 trees using j - 1 numbers and n -j numbers.  
   So we add number(j - 1) \* number(n - j) to our solution.

**Code -**

class Solution:

# @param A : integer

# @return an integer

def numTrees(self, A):

if(A<=2): return A

dp = [0 for i in range(A+1)]

dp[0] = 1

dp[1] = 1

dp[2] = 2

for i in range(3, A+1):

dp[i] = 0

for j in range(i):

dp[i] += (dp[j]\*dp[i-j-1])

return dp[A]

**Q32. Intersecting Chords in a Circle**

Given a number **A**, return number of ways you can draw **A** chords in a circle with **2 x A** points such that no **2** chords intersect.

Two ways are different if there exists a chord which is present in one way and not in other.

Return the answer modulo **109 + 7**.

**Logic -**

Can we relate answer for N with smaller answers.

If we draw a chord between any two points, can you observe current set of points getting broken into two smaller sets S\_1 and S\_2? Can a chord be drawn between two points where each point belong to different set?

If we draw a chord from a point in S\_1 to a point in S\_2, it will surely intersect the chord we’ve just drawn.

So, we can arrive at a recurrence that Ways(n) = sum[i = 0 to n-1] { Ways(i)\*Ways(n-i-1) }.

Here we iterate over i, assuming that size of one of the sets is i and size of other set automatically is (n-i-1) since we’ve already used a pair of points and i pair of points in one set.

**Code -**

public class Solution {

public int chordCnt(int A) {

if(A<=2) return A;

int mod = (1000\*1000\*1000+7);

long[] dp = new long[A+1];

Arrays.fill(dp, 0);

dp[0] = 1;

dp[1] = 1;

dp[2] = 2;

for(int i=3;i<=A;i++) {

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

dp[i] += (dp[j]\*dp[i-j-1]);

dp[i] %= mod;

}

}

return (int)(dp[A] % mod);

}

}

**Q33. Best Time to Buy and Sell Stocks I**

Say you have an array, **A**, for which the **i**th element is the price of a given stock on day **i**.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Return the **maximum** possible profit.

**Logic –** Maintain a max variable from right. We consider each point as a buying point and the max variable as a selling point and find the maximum profit.

**Code -**

class Solution:

def maxProfit(self, A):

if(len(A)==0): return 0

maxR = A[-1]

ans = 0

for i in range(len(A)-2, -1, -1):

ans = max(ans, maxR - A[i])

maxR = max(maxR, A[i])

return ans

**Q34. Best Time to Buy and Sell Stocks II**

Say you have an array, **A**, for which the **ith** element is the price of a given stock on day **i**.

Design an algorithm to find the maximum profit.

You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

**Logic -**

Observation based solution:

Note 1: I will never buy a stock and sell it in loss.

Note 2: If A[i] < A[i+1], I will always buy a stock on i and sell it on i+1.  
Think and try to come up with a proof on the validity of the statement.

**DP solution**

We can choose between buy/sell or doing nothing for that day. If we have bought a stock we can sell it and if not bought we can buy

**Code -**

public class Solution {

// DO NOT MODIFY THE LIST. IT IS READ ONLY

public int maxProfit(final List<Integer> A) {

int [][] dp = new int[A.size()+1][2];

for(int i=A.size()-1;i>=0;i--) {

for(int j=0;j<2;j++) {

if(j==0) {

// buy

dp[i][j] = Math.max(dp[i+1][0], dp[i+1][1]-A.get(i));

}

else {

// sell

dp[i][j] = Math.max(dp[i+1][1], dp[i+1][0]+A.get(i));

}

}

}

return dp[0][0];

}

}

**Q35. Best Time to Buy and Sell Stocks III**

Say you have an array, **A**, for which the **ith** element is the price of a given stock on day **i**.

Design an algorithm to find the **maximum** profit. You may complete at most **2** transactions.

Return the maximum possible profit.

**Note:** You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

**Logic -**

**Same as above problem**

We can choose between buy/sell or doing nothing for that day. If we have bought a stock we can sell it and if not bought we can buy

After selling a stock we can update the number of transactions done, if transactions become equal to 2 then we will stop the process right there and compare next elements for max profit.

**Code -**

public class Solution {

public int maxProfit(final List<Integer> A) {

int [][][] dp = new int[A.size()+1][2][3];

for(int i=A.size()-1;i>=0;i--) {

for(int j=0;j<2;j++) {

for(int k=1;k<=2;k++) {

if(j==0) {

// buy

dp[i][j][k] = Math.max(dp[i+1][0][k], dp[i+1][1][k]-A.get(i));

}

else {

// sell

dp[i][j][k] = Math.max(dp[i+1][1][k], dp[i+1][0][k-1]+A.get(i));

}

}

}

}

return dp[0][0][2];

}

}

**Q 22 and 27 is left**