**0-1 Knapsack Problem :-** Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).

**Method 1:** **Overlapping Subproblems :-** Following is recursive implementation that simply follows the recursive structure mentioned above.

class Knapsack

{

    // A utility function that returns maximum of two integers

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

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

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

     {

        // Base Case

    if (n == 0 || W == 0)

        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then

    // this item cannot be included in the optimal solution

    if (wt[n-1] > W)

       return knapSack(W, wt, val, n-1);

// Return the maximum of two cases:(1) nth item included (2) not included

    else

return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1), knapSack(W, wt, val, n-1));

      }

   public static void main(String args[])

   {

    int val[] = new int[]{60, 100, 120},wt[] = new int[]{10, 20, 30};

    int  W = 50, n = val.length;

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

    }

}

Output:- 220, It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

In the following recursion tree, K() refers to knapSack(). The two parameters indicated in the following recursion tree are n and W. The recursion tree is for following sample inputs.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

K(3, 2) ---------> K(n, W)

/ \

K(2,2) K(2,1)

/ \ / \

K(1,2) K(1,1) K(1,1) K(1,0)

/ \ / \ / \

K(0,2) K(0,1) K(0,1) K(0,0) K(0,1) K(0,0)

Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight. Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](http://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

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

class Knapsack

{

    // A utility function that returns maximum of two integers

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

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

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

    {

         int i, w;

     int K[][] = new int[n+1][W+1];

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

     for (i = 0; i <= n; i++)

     {

         for (w = 0; w <= W; w++)

         {

             if (i==0 || w==0)

                  K[i][w] = 0;

             else if (wt[i-1] <= w)

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

             else

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

         }

      }

      return K[n][W];

    }

public static void main(String args[])

    {

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

    int wt[] = new int[]{10, 20, 30};

    int  W = 50;

    int n = val.length;

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

    }

}

### Asked:[Amazon](http://practice.geeksforgeeks.org/company/Amazon/), [Flipkart](http://practice.geeksforgeeks.org/company/Flipkart/), [GreyOrange](http://practice.geeksforgeeks.org/company/GreyOrange%20/), [Microsoft](http://practice.geeksforgeeks.org/company/Microsoft/), [Mobicip](http://practice.geeksforgeeks.org/company/Mobicip/), [MorganStanley](http://practice.geeksforgeeks.org/company/Morgan-Stanley/), [Oracle](http://practice.geeksforgeeks.org/company/Oracle/), [Payu](http://practice.geeksforgeeks.org/company/Payu/), [Snapdeal](http://practice.geeksforgeeks.org/company/Snapdeal/), [Visa](http://practice.geeksforgeeks.org/company/Visa/)

# Maximum Sum Increasing Subsequence:- Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the intgers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be 106 (1 + 2 + 3 + 100), if the input array is {3, 4, 5, 10}, then output should be 22 (3 + 4 + 5 + 10) and if the input array is {10, 5, 4, 3}, then output should be 10.

public class MaximumSumSubsequence {

public int maxSum(int arr[]){

int T[] = new int[arr.length];

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

T[i] = arr[i];

}

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

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

if(arr[j] < arr[i]){

T[i] = Math.max(T[i], T[j] + arr[i]);

}

}

}

int max = T[0];

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

if(T[i] > max){

max = T[i];

}

}

return max;

}

public static void main(String args[]){

MaximumSumSubsequence mss = new MaximumSumSubsequence();

int arr[] = {1, 101, 10, 2, 3, 100,4};

int r = mss.maxSum(arr);

System.out.print(r);

}

}

**Note:-**This is maximum sum increasing subsequence. This is different from longest subsequence. For the array 4 6 1 3 8 4 6 longest sub sequence is 1 3 4 6.And the maximum sum increasing is 4 6 8.

# Find maximum subsquare in a matrix made up of Xs and Os such that all four sides of subsquare are Xs. It does not matter what is inside the subsquare. All 4 sides should be made up entirely of Xs

\* 0 0 0 0 0 X 0,0 0,0 0,0 0,0 0,0 1,1

\* 0 X 0 X X X 0,0 1,1 0,0 1,1 1,2 2,3

\* 0 X 0 X 0 X 0,0 2,1 0,0 2,1 0,0 3,1

\* 0 X X X X X 0,0 3,1 1,2 3,3 1,4 4,5

\* 0 0 0 0 0 0 0,0 0,0 0,0 0,0 0,0 0,0

\*

\* Output of above program should be 3

\*

**\* Solution:-** Have another matrix which is capable of holding 2 values hori and ver.Ver stores how far vertically you can see Xs. Hori stores how far horizontally you can see Xs.

Once this matrix is build look for biggest subsquare by getting min of hori and ver at each point and checking if subsquare can be formed from value min to 1.

\* Test cases:

\* Matrix entirely made up of Xs

\* Matrix entirely made up of Os

\* Matrix with Xs and Os but maximum subsquare is length 1

\*/

public class SubsquareSurrounedByXs {

class Cell{

int ver;

int hori;

}

public int findSubSquare(char input[][]){

Cell T[][] = new Cell[input.length][input[0].length];

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

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

T[i][j] = new Cell();

}

}

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

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

if(input[i][j] == 'X'){

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

T[i][j].hori = 1;

T[i][j].ver = 1;

}

else if(i == 0){

T[i][j].hori = T[i][j-1].hori + 1;

T[i][j].ver = 1;

}else if(j == 0){

T[i][j].ver = T[i-1][j].ver +1;

T[i][j].hori = 1;

}else{

T[i][j].hori = T[i][j-1].hori +1;

T[i][j].ver = T[i-1][j].ver + 1;

}

}

}

}

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

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

System.out.print(T[i][j].ver + "," + T[i][j].hori+ " ");

}

System.out.println();

}

//start iterating from bottom right corner and find min of hori or ver at every cell.

//If this is greater than 1 then see if you can find a number between this min and 1

//such that on left's ver and top's hori is greater greater than or equal to k.

int max = 1;

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

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

if(T[i][j].ver == 0 || T[i][j].ver == 1 || T[i][j].hori ==1 ){

continue;

}

int min = Math.min(T[i][j].ver, T[i][j].hori);

int k = 0;

for(k=min; k > 1; k--){

if(T[i][j-k+1].ver >= k && T[i-k+1][j].hori >= k){

break;

}

}

if(max < k){

max = k;

}

}

}

return max;

}

public static void main(String args[]){

char [][] input1 = {{'O', 'O', 'O', 'O', 'O', 'X'},

{'O', 'X', 'O', 'X', 'X', 'X'},

{'O', 'X', 'O', 'X', 'O', 'X'},

{'O', 'X', 'X', 'X', 'X', 'X'},

{'O', 'O', 'O', 'O', 'O', 'O'},

};

SubsquareSurrounedByXs ss = new SubsquareSurrounedByXs();

System.out.println(ss.findSubSquare(input));

}

}

**Given a preorder sequence how many unique trees can be createdSolution is catalan number. Number of tree is exactly same as number of unique BST create with array of size n.**

The way it works for preorder sequence is as follows

Suppose our preorder sequence is 1 2 3 4 So we need to compute following things

count(3)\* 2 (combination of 2,3 and 4 on both side of 1) count(1)\*count(2) (combination of 2 on one side and 3, 4 on other side) count(2)\*count(1) (combinatino of 2,3 on one side and 4 on other side) count(3)\*2 can be broken into count(3)\*count(0) + count(0)\*count(3)

\* So our final result is

count(0)\*count(3) + count(1)\*count(2) + count(2)\*count(1) + count(3)\*count(0)

\* which is a catalan number \*/

public class CountNumberOfTreePreorder {

public int count(int num){

if(num == 0){

return 0;

}

int T[] = new int[num+1];

T[0] = 1;

T[1] = 1;

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

int sum = 0;

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

sum += T[k]\*T[i-k-1];

}

T[i] = sum;

}

return T[num];

}

public int countRec(int num){

if(num == 0 || num ==1){

return 1;

}

int sum = 0;

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

sum += countRec(i-1)\*countRec(num-i);

}

return sum;

}

public static void main(String args[]){

CountNumberOfTreePreorder cn = new CountNumberOfTreePreorder();

System.out.println(cn.count(3));

System.out.println(cn.count(4));

System.out.println(cn.count(5));

System.out.println(cn.countRec(3));

System.out.println(cn.countRec(4));

System.out.println(cn.countRec(5));

}

}

# How to print maximum number of A’s using given four keys:- This is a famous interview question asked in [Google](http://www.careercup.com/question?id=7184083), [Paytm](http://www.geeksforgeeks.org/one97paytm-interview-experience/) and many other company interviews.

# Below is the problem statement. Imagine you have a special keyboard with the following keys:

# Key 1: Prints 'A' on screen

# Key 2: (Ctrl-A): Select screen

# Key 3: (Ctrl-C): Copy selection to buffer

# Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

# If you can only press the keyboard for N times (with the above four keys), write a program to produce maximum numbers of A's. That is to say, the input parameter is N (No. of keys that you can press), the output is M (No. of As that you can produce).

# Input: N = 3 Output: 3

# We can at most get 3 A's on screen by pressing following key sequence. A, A, A

# Input: N = 7 Output: 9

# We can at most get 9 A's on screen by pressing following key sequence.

# A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

# Input: N = 11 Output: 27

# We can at most get 27 A's on screen by pressing following key sequence.

# A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V, Ctrl A, Ctrl C, Ctrl V, Ctrl V

# Below are few important points to note. a) For N < 7, the output is N itself. b) Ctrl V can be used multiple times to print current buffer (See last two examples above). The idea is to compute the optimal string length for N keystrokes by using a simple insight. The sequence of N keystrokes which produces an optimal string length will end with a suffix of Ctrl-A, a Ctrl-C, followed by only Ctrl-V's (For N > 6). The task is to find out the break=point after which we get the above suffix of keystrokes. Definition of a breakpoint is that instance after which we need to only press Ctrl-A, Ctrl-C once and the only Ctrl-V’s afterwards to generate the optimal length. If we loop from N-3 to 1 and choose each of these values for the break-point, and compute that optimal string they would produce. Once the loop ends, we will have the maximum of the optimal lengths for various breakpoints, thereby giving us the optimal length for N keystrokes.

# Below is Dynamic Programming based C implementation where an auxiliary array screen[N] is used to store result of subproblems.

// this function returns the optimal length string for N keystrokes

int findoptimal(int N)

{

    // The optimal string length is N when N is smaller than 7

    if (N <= 6)

        return N;

    // An array to store result of subproblems

    int screen[N];

    int b;  // To pick a breakpoint

    // Initializing the optimal lengths array for uptil 6 input strokes.

    int n;

    for (n=1; n<=6; n++)

        screen[n-1] = n;

    // Solve all subproblems in bottom manner

    for (n=7; n<=N; n++)

    {

        // Initialize length of optimal string for n keystrokes

        screen[n-1] = 0;

        // For any keystroke n, we need to loop from n-3 keystrokes

        // back to 1 keystroke to find a breakpoint 'b' after which we

        // will have ctrl-a, ctrl-c and then only ctrl-v all the way.

        for (b=n-3; b>=1; b--)

        {

            // if the breakpoint is at b'th keystroke then

            // the optimal string would have length (n-b-1)\*screen[b-1];

            int curr = (n-b-1)\*screen[b-1];

            if (curr > screen[n-1])

                screen[n-1] = curr;

        }

    }

    return screen[N-1];

}

int main()

{

    int N;

//for the rest of the array we will rely on the previous entries to compute new ones

for (N=1; N<=20; N++)

 printf("Maximum Number of A's with %d keystrokes is %d\n",N,findoptimal(N));

}

# Output:-

# Maximum Number of A's with 1 keystrokes is 1

# Maximum Number of A's with 2 keystrokes is 2

# ………..

# ………….

# Maximum Number of A's with 19 keystrokes is 256

# Maximum Number of A's with 20 keystrokes is 324

**Minimum number of jumps to reach end :-** Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then cannot move through that element.

Example:- Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}

Output: 3 (1-> 3 -> 8 ->9)

First element is 1, so can only go to 3. Second element is 3, so can make at most 3 steps eg to 5 or 8 or 9. In this method, we build a jumps[] array from left to right such that jumps[i] indicates the minimum number of jumps needed to reach arr[i] from arr[0]. Finally, we return jumps[n-1].

public class MinJumpToReachEnd {

public int minJump(int arr[],int result[]){

int []jump = new int[arr.length];

jump[0] = 0;

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

jump[i] = Integer.MAX\_VALUE-1;

}

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

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

if(arr[j] + j >= i){

if(jump[i] > jump[j] + 1){

result[i] = j;

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

}

}

}

}

return jump[jump.length-1];

}

public int jump(int[] nums) {

if (nums.length == 1) {

return 0;

}

int count = 0;

int i = 0;

while (i + nums[i] < nums.length - 1) {

int maxVal = 0;

int maxValIndex = 0;

for (int j = 1; j <= nums[i]; j++) {

if (nums[j + i] + j > maxVal) {

maxVal = nums[j + i] + j;

maxValIndex = i + j;

}

}

i = maxValIndex;

count++;

}

return count + 1;

}

public static void main(String args[]){

MinJumpToReachEnd mj = new MinJumpToReachEnd();

int arr[] = {1,3,5,3,2,2,6,1,6,8,9};

int r[] = new int[arr.length];

int result = mj.minJump(arr,r);

System.out.println(result);

int i = arr.length-1;

Arrays.toString(r);

int arr1[] = {2,3,1,1,4};

System.out.print(mj.jump(arr));

}

}

Space complexity O(n) to maintain result and min jumps

\* Time complexity O(n^2)

**Subset Sum Problem:-** Given a set of non-negative integers, and a value sum, determine if there is a subset of the given set with sum equal to given sum.

Examples: set[] = {3, 34, 4, 12, 5, 2}, sum = 9

Output: True //There is a subset (4, 5) with sum 9.

\* Given an array of non negative numbers and a total, is there subset of numbers in this array which adds up

\* to given total. Another variation is given an array is it possible to split it up into 2 equal

\* sum partitions. Partition need not be equal sized. Just equal sum.

public class SubsetSum {

public boolean subsetSum(int input[], int total) {

boolean T[][] = new boolean[input.length + 1][total + 1];

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

T[i][0] = true;

}

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

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

if (j - input[i - 1] >= 0) {

T[i][j] = T[i - 1][j] || T[i - 1][j - input[i - 1]];

} else {

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

}

}

}

return T[input.length][total];

}

public boolean partition(int arr[]) {

int sum = 0;

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

sum += arr[i];

}

if (sum % 2 != 0) {

return false;

}

sum = sum / 2;

boolean[][] T = new boolean[arr.length + 1][sum + 1];

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

T[i][0] = true;

}

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

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

if (j - arr[i - 1] >= 0) {

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

} else {

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

}

}

}

return T[arr.length][sum];

}

public static void main(String args[]) {

SubsetSum ss = new SubsetSum();

int arr[] = {1, 3, 5, 5, 2, 1, 1, 6};

System.out.println(ss.partition(arr));

int arr1[] = {2, 3, 7, 8};

System.out.print(ss.subsetSum(arr1, 11));

}

}

# Minimum Cost Path Dynamic Programming:-

public class MinCostPath {

public int minCost(int [][]cost,int m,int n){

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

int sum = 0;

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

temp[0][i] = sum + cost[0][i];

sum = temp[0][i];

}

sum = 0;

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

temp[i][0] = sum + cost[i][0];

sum = temp[i][0];

}

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

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

temp[i][j] = cost[i][j] + min(temp[i-1][j-1], temp[i-1][j],temp[i][j-1]);

}

}

return temp[m][n];

}

public static void main(String args[]){

MinCostPath mcp = new MinCostPath();

int cost[][] = {{1,2,3},{4,8,2},{1,5,3},{6,2,9}};

int result = mcp.minCost(cost, 3, 2);

int result1 = mcp.minCostRec(cost, 3, 2);

System.out.println(result);

System.out.println(result1);

}

}

# Cutting a Rod:-Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n.Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

# length | 1 2 3 4 5 6 7 8

# --------------------------------------------

# price | 1 5 8 9 10 17 17 20

public int maxValue(int price[]){

int max[] = new int[price.length+1];

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

for(int j=i; j <= price.length; j++){

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

}

}

return max[price.length];

}

public static void main(String args[]){

CuttingRod cr =new CuttingRod();

int[] price = {3,5,8,9,10,20,22,25};

long t1 = System.currentTimeMillis();

int r = cr.recursiveMaxValue(price,8);

long t2 = System.currentTimeMillis();

System.out.println(r);

System.out.println(t2 - t1);

}

}

Partition a set into two subsets such that the difference of subset sums is minimum

Given a set of integers, the task is to divide it into two sets S1 and S2 such that the absolute difference between their sums is minimum.

If there is a set S with n elements, then if we assume Subset1 has m elements, Subset2 must have n-m elements and the value of abs(sum(Subset1) – sum(Subset2)) should be minimum.

Input: arr[] = {1, 6, 11, 5} Output: 1

Explanation:- Subset1 = {1, 5, 6}, sum of Subset1 = 12

Subset2 = {11}, sum of Subset2 = 11

// Returns the minimum value of the difference of the two sets.

int findMin(int arr[], int n)

{

    // Calculate sum of all elements

    int sum = 0;

    for (int i = 0; i < n; i++)

        sum += arr[i];

    // Create an array to store results of subproblems

    bool dp[n+1][sum+1];

    // Initialize first column as true. 0 sum is possible with all elements.

    for (int i = 0; i <= n; i++)

        dp[i][0] = true;

//Initialize top row, except dp[0][0], as false. With 0 elements, no other sum except 0 is possible

    for (int i = 1; i <= sum; i++)

        dp[0][i] = false;

    // Fill the partition table in bottom up manner

    for (int i=1; i<=n; i++)

    {

        for (int j=1; j<=sum; j++)

        {

            // If i'th element is excluded

            dp[i][j] = dp[i-1][j];

            // If i'th element is included

            if (arr[i-1] <= j)

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

        }

    }

    // Initialize difference of two sums.

    int diff = INT\_MAX;

//Find the largest j such that dp[n][j] is true where j loops from sum/2 t0 0

    for (int j=sum/2; j>=0; j--)

    {

        // Find the

        if (dp[n][j] == true)

        {

            diff = sum-2\*j;

            break;

        }

    }

    return diff;

}

int main()

{

    int arr[] = {3, 1, 4, 2, 2, 1};

    int n = sizeof(arr)/sizeof(arr[0]);

    cout << "The minimum difference between 2 sets is "

         << findMin(arr, n);

    return 0;

}

**Output**:-The minimum difference between 2 sets is 1

Time Complexity = O(n\*sum) where n is number of elements and sum is sum of all elements.

# Egg Dropping Puzzle:-The following is a description of the instance of this famous puzzle involving n=2 eggs and a building with k=36 floors.

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

…..An egg that survives a fall can be used again.  
…..A broken egg must be discarded.  
…..The effect of a fall is the same for all eggs.  
…..If an egg breaks when dropped, then it would break if dropped from a higher floor.  
…..If an egg survives a fall then it would survive a shorter fall.  
…..It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?  
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

public class EggDropping {

public int calculate(int eggs, int floors){

int T[][] = new int[eggs+1][floors+1];

int c =0;

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

T[1][i] = i;

}

for(int e = 2; e <= eggs; e++){

for(int f = 1; f <=floors; f++){

T[e][f] = Integer.MAX\_VALUE;

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

c = 1 + Math.max(T[e-1][k-1], T[e][f-k]);

if(c < T[e][f]){

T[e][f] = c;

}

}

}

}

return T[eggs][floors];

}

public static void main(String args[]){

EggDropping ed = new EggDropping();

int r = ed.calculate(3,100);

System.out.println(r);

}

}