Egg Dropping Puzzle | DP-11 - GeeksforGeeks

Source: https://www.geeksforgeeks.org/egg-dropping-puzzle-dp-11/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Egg Dropping Puzzle | DP-11 Last Updated : 3 Nov, 2025 You are given n identical eggs and you have access to a k -floored building from 1 to k. There exists a floor f where 0 <= f <= k such that any egg dropped from a floor higher than f will break, and any egg dropped from or below floor f will not break . There are a few rules given below: 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 the egg doesn't break at a certain floor, it will not break at any floor below. If the egg breaks on a certain floor, it will break on any floor above. Your task is to find the minimum number of moves you need to determine the value of f with certainty. Example: Input: n = 2, k = 36 Output: 8 Explanation: In all the situations, 8 maximum moves are required to find the maximum floor. Following is the strategy to do so: Drop from floor 8 → If breaks, check 1-7 sequentially. Drop from floor 15 → If breaks, check 9-14. Drop from floor 21 → If breaks, check 16-20. Drop from floor 26 → If breaks, check 22-25. Drop from floor 30 → If breaks, check 27-29. Drop from floor 33 → If breaks, check 31-32. Drop from floor 35 → If breaks, check 34. Drop from floor 36 → Final check. Input: n = 1, k = 36 Output: 36 Explanation: 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. Input: n = 2, k = 10 Output: 4 Explanation: In all the situations, 4 maximum moves are required to find the maximum floor. Following is the strategy to do so: Drop from floor 4 → If breaks, check 1-3 sequentially. Drop from floor 7 → If breaks, check 5-6. Drop from floor 9 → If breaks, check 8. Drop from floor 10 → Final check. Try it on GfG Practice Table of Content [Naive Approach] Using Recursion - O(n ^ k) Time and O(1) Space Analyzing the Overlapping Subproblems in Egg Dropping Puzzle [Better Approach] Using Memoization - O(n * k^2) Time and O(n * k) Space [Expected Approach] Using Tabulation with Optimization - O(n * k) Time and O(n * k) Space [Optimized Approach] Using Space-Optimized Table - O(n * k) Time and O(n) Space [Naive Approach] Using Recursion - O(n ^ k) Time and O(1) Space The idea is to try dropping an egg from every floor (from 1 to K) and recursively calculate the minimum number of droppings needed in the worst case. To do so, run a loop from i equal to 1 to K, where i denotes the current floor. When we drop an egg from floor i, there are two possibilities: If the egg breaks: Then we need to check for floors lower than i with 1 less egg, i.e. the value of both egg and floor will be reduced by 1. If the egg doesn't break: Then we need to check for floors higher than i with same number of eggs, i.e. the number of floors to check will be k - i, and the count of eggs will remain same. The maximum of these two move + 1(current move) will be considered, and the minimum of this is our answer. For each recursive call, follow the below given steps: If k == 1 or k == 0 then return k, i.e. we need to check only 1 or no floors. If n == 1 return k, i.e. we need to check all floors starting from the first one. Create an integer res equal to the integer Maximum. Run a for loop from i equal to 1 till i is less than or equal to k. Set curr equal to maximum of eggDrop(n-1, i-1) or eggDrop(n, k-i). If curr is less than res then set res equal to curr. Return res C++ #include <bits/stdc++.h> using namespace std ; // Function to find minimum number of attempts // needed in order to find the critical floor int eggDrop (int n , int k) { // if there is less than or equal to one floor if (k == 1 || k == 0) return k ; // if there is only one egg if (n == 1) return k ; // to store the minimum number of attempts int res = INT_MAX ; // Consider all droppings from // 1st floor to kth floor for (int i = 1 ; i <= k ; i ++) { int cur = 1 + max (eggDrop (n - 1 , i - 1), eggDrop (n , k - i)); if (cur < res) res = cur ; } return res ; } int main () { int n = 2 , k = 10 ; cout << eggDrop (n , k); return 0 ; } Java // Function to find minimum number of attempts // needed in order to find the critical floor import java.util.* ; class GfG

{ // Function to find minimum number of attempts // needed in order to find the critical floor static int eggDrop (int n , int k) { // if there is less than or equal to one floor if (k == 1 || k == 0) return k ; // if there is only one egg if (n == 1) return k ; // to store the minimum number of attempts int res = Integer . MAX_VALUE ; // Consider all droppings from // 1st floor to kth floor for (int i = 1 ; i <= k ; i ++) { int cur = 1 + Math . max (eggDrop (n - 1 , i - 1), eggDrop (n , k - i)); if (cur < res) res = cur ; } return res ; } public static void main (String [] args) { int n = 2 , k = 10 ; System . out . println (eggDrop (n , k)); } } Python # Function to find minimum number of attempts # needed in order to find the critical floor def eggDrop (n , k): # if there is less than or equal to one floor if k == 1 or k == 0 : return k # if there is only one egg if n == 1 : return k # to store the minimum number of attempts res = float ('inf') # Consider all droppings from # 1st floor to kth floor for i in range (1 , k + 1): cur = 1 + max (eggDrop (n - 1 , i - 1), eggDrop (n , k - i)) if cur < res : res = cur return res if __name__ == "__main__" : n = 2 k = 10 print (eggDrop (n , k)) C# // Function to find minimum number of attempts // needed in order to find the critical floor using System ; class GfG { // Function to find minimum number of attempts // needed in order to find the critical floor static int eggDrop (int n , int k) { // if there is less than or equal to one floor if (k == 1 || k == 0) return k ; // if there is only one egg if (n == 1) return k ; // to store the minimum number of attempts int res = int . MaxValue ; // Consider all droppings from // 1st floor to kth floor for (int i = 1 ; i <= k ; i ++) { int cur = 1 + Math . Max (eggDrop (n - 1 , i - 1), eggDrop (n , k - i)); if (cur < res) res = cur ; } return res ; } static void Main () { int n = 2 , k = 10 ; Console . WriteLine (eggDrop (n , k)); } } JavaScript // Function to find minimum number of attempts // needed in order to find the critical floor function eggDrop (n , k) { // if there is less than or equal to one floor if (k === 1 || k === 0) return k ; // if there is only one egg if (n === 1) return k ; // to store the minimum number of attempts let res = Number . MAX_SAFE_INTEGER ; // Consider all droppings from // 1st floor to kth floor for (let i = 1 ; i <= k ; i ++) { let cur = 1 + Math . max (eggDrop (n - 1 , i - 1), eggDrop (n , k - i)); if (cur < res) res = cur ; } return res ; } let n = 2 , k = 10 ; console . log (eggDrop (n , k)); Output 4 Analyzing the Overlapping Subproblems in Egg Dropping Puzzle In the recursive solution, many subproblems are computed more than once. Consider calculating the number of attempts for 2 eggs and 10 floors. When dropping an egg from a certain floor i , the problem divides into two subproblems: one for the floors below (eggDrop(1, i-1)) one for the floors above (eggDrop(2, 10-i)). Different choices of i can lead to the same subproblem being solved multiple times (for example, eggDrop(2, 3) might be computed in several different branches). This repeated computation of the same subproblems demonstrates overlapping subproblems , which makes memoization a highly effective optimisation technique. [Better Approach] Using Memoization - O(n * k^2) Time and O(n * k) Space The above approach can be optimized using memoization , as we are computing the same sub-problem multiple times. The idea is to create a 2d array memo[][] of order n * k, to store the results of the sub-problem, where memo[i][j] stores the result of i eggs and j floors . For each recursive call, check if the value is already computed, if so, return the stored value, else proceed similar to the above approach, and at last store the results in memo[][] and return the value. C++ #include <bits/stdc++.h> using namespace std ; // Function to find minimum number of attempts // needed in order to find the critical floor int solveEggDrop (int n , int k , vector < vector < int >> & memo) { // if value is already calculated if (memo [n][k] != -1) { return memo [n][k]; } // if there is less than or equal to one floor if (k == 1 || k == 0) return k ; // if there is only one egg if (n == 1) return k ; // to store the minimum number of attempts int res = INT_MAX ; // Consider all droppings from // 1st floor to kth floor for (int i = 1 ; i <= k ; i ++) { int cur = max (solveEggDrop (n - 1 , i - 1 , memo), solveEggDrop (n , k - i , memo)); if (cur < res) res = cur ; } // update the memo, and return return memo [n][k] = res + 1 ; } // Function to find minimum number of attempts // needed in order to find the critical floor int eggDrop (int n , int k) { // create memo table vector < vector < int >> memo (n + 1 , vector < int > (k + 1 , -1)); return solveEggDrop (n , k , memo); } int main () { int n = 2 , k = 10 ; cout << eggDrop (n , k); return 0 ; } Java // Function to find minimum number of attempts // needed in order to find the critical floor import java.util.* ; class GfG { // Function to find minimum number of attempts // needed in order to find the critical floor static int solveEggDrop (int n , int k , int [][] memo) { // if value is already calculated if (memo [n][k] != - 1) { return memo [n][k] ; } // if there is less than or equal to one floor if (k == 1 || k == 0) return k ; // if there is only one egg if (n == 1) return k ; // to store the minimum number of attempts int res = Integer . MAX_VALUE ; // Consider all droppings from // 1st floor to kth floor for (int i = 1 ; i <= k ; i ++) { int cur = Math . max (solveEggDrop (n - 1 , i - 1 , memo), solveEggDrop (n , k - i , memo)); if (cur < res) res = cur ; } // update the memo, and return memo [n][k] = res + 1 ; return memo [n][k] ; } // Function to find minimum number of attempts // needed in order to find the critical floor static int eggDrop (int n , int k) { // create memo table int [][] memo = new int [n + 1][k + 1] ; for (int i = 0 ; i <= n ; i ++) { Arrays . fill (

memo [i] , - 1); } return solveEggDrop (n , k , memo); } public static void main (String [] args) { int n = 2 , k = 10 ; System . out . println (eggDrop (n , k)); } } Python # Function to find minimum number of attempts # needed in order to find the critical floor def solveEggDrop (n , k , memo): # if value is already calculated if memo [n][k] != - 1 : return memo [n][k] # if there is less than or equal to one floor if k == 1 or k == 0 : return k # if there is only one egg if n == 1 : return k # to store the minimum number of attempts res = float ('inf') # Consider all droppings from # 1st floor to kth floor for i in range (1 , k + 1): cur = max (solveEggDrop (n - 1 , i - 1 , memo), \ solveEggDrop (n , k - i , memo)) if cur < res : res = cur # update the memo, and return memo [n][k] = res + 1 return memo [n][k] # Function to find minimum number of attempts # needed in order to find the critical floor def eggDrop (n , k): # create memo table memo = [[- 1 for _ in range (k + 1)] for _ in range (n + 1)] return solveEggDrop (n , k , memo) if __name__ == "__main__" : n = 2 k = 10 print (eggDrop (n , k)) C# // Function to find minimum number of attempts // needed in order to find the critical floor using System ; class GfG { // Function to find minimum number of attempts // needed in order to find the critical floor static int solveEggDrop (int n , int k , int [][] memo) { // if value is already calculated if (memo [n][k] != - 1) return memo [n][k]; // if there is less than or equal to one floor if (k == 1 || k == 0) return k ; // if there is only one egg if (n == 1) return k ; // to store the minimum number of attempts int res = int . MaxValue ; // Consider all droppings from // 1st floor to kth floor for (int i = 1 ; i <= k ; i ++) { int cur = Math . Max (solveEggDrop (n - 1 , i - 1 , memo), solveEggDrop (n , k - i , memo)); if (cur < res) res = cur ; } // update the memo, and return memo [n][k] = res + 1 ; return memo [n][k]; } // Function to find minimum number of attempts // needed in order to find the critical floor static int eggDrop (int n , int k) { // create memo table int [][] memo = new int [n + 1][]; for (int i = 0 ; i <= n ; i ++) { memo [i] = new int [k + 1]; for (int j = 0 ; j <= k ; j ++) { memo [i][j] = - 1 ; } } return solveEggDrop (n , k , memo); } static void Main () { int n = 2 , k = 10 ; Console . WriteLine (eggDrop (n , k)); } } JavaScript // Function to find minimum number of attempts // needed in order to find the critical floor function solveEggDrop (n , k , memo) { // if value is already calculated if (memo [n][k] !== - 1) return memo [n][k]; // if there is less than or equal to one floor if (k === 1 || k === 0) return k ; // if there is only one egg if (n === 1) return k ; // to store the minimum number of attempts let res = Number . MAX_SAFE_INTEGER ; // Consider all droppings from // 1st floor to kth floor for (let i = 1 ; i <= k ; i ++) { let cur = Math . max (solveEggDrop (n - 1 , i - 1 , memo), solveEggDrop (n , k - i , memo)); if (cur < res) res = cur ; } // update the memo, and return memo [n][k] = res + 1 ; return memo [n][k]; } // Function to find minimum number of attempts // needed in order to find the critical floor function eggDrop (n , k) { // create memo table let memo = new Array (n + 1); for (let i = 0 ; i <= n ; i ++) { memo [i] = new Array (k + 1). fill (- 1); } return solveEggDrop (n , k , memo); } // Driver code let n = 2 , k = 10 ; console . log (eggDrop (n , k)); Output 4 [Expected Approach] Using Tabulation with Optimization - O(n * k) Time and O(n * k) Space A direct tabulation based solution for the above memoization and recursive approach would require O(n * k^2) Time and O(n * k) Space. We can further optimize the above approach. In the above approach, each state memo[i][j] stores the number of moves required to solve the sub-problem of i eggs and j floors . Instead of doing so, the idea is to make dp[i][j] store the maximum number of floors that can be processed using i moves and j eggs. By doing so, we will only be required to find the maximum reachable floor in the previous move, and will avoid checking each k floor for all the states. Step-by-step approach: Create a 2d table dp[][] of order k * n (as k is the maximum possible moves required). Initialize the counter cnt to store the number of moves. Run a while loop until the number of floors i.e. dp[cnt][n] is less than k. At each iteration increment the moves by 1 (i.e. increment cnt by 1), and run a loop from 1 to n, representing the number of eggs. At each iteration, there are two possibilities: If egg breaks, then check the result of dp[cnt - 1][i - 1]. If egg doesn't break, then check the result of dp[cnt - 1][i]. Update dp[cnt][i] = 1 + dp[cnt - 1][i - 1] + dp[cnt - 1][i], where we are adding 1 for the current floor. At last return the count of moves cnt. C++ #include <bits/stdc++.h> using namespace std ; // Function to find minimum number of attempts // needed in order to find the critical floor int eggDrop (int n , int k) { // create a 2D table to store the results vector < vector < int >> dp (k + 1 , vector < int > (n + 1 , 0)); // to count the number of moves int cnt = 0 ; // while the number of floors is less than k while (dp [cnt][n] < k) { cnt ++ ; // for each egg for (int i = 1 ; i <= n ; i ++) { dp [cnt][i] = 1 + dp [cnt - 1][i - 1] + dp [cnt - 1][i]; } } return cnt ; } int main () { int n = 2 , k = 10 ; cout << eggDrop (n , k); return 0 ; } Java // Function to find minimum number of attempts // needed in order to find the critical floor import java.util.* ; class GfG { // Function to find minimum number of attempts // needed in order to find the critical floor static int eggDrop (int n , int k) { // create a 2D table to store the results int [][] dp = new int [k + 1][n + 1] ; // to count the number of moves int cnt = 0 ; // while the number of floors is less than k while (dp [cnt][n] < k) { cnt ++ ; // for each egg for

(int i = 1 ; i <= n ; i ++) { dp [cnt][i] = 1 + dp [cnt - 1][i - 1] + dp [cnt - 1][i] ; } } return cnt ; } public static void main (String [] args) { int n = 2 , k = 10 ; System . out . println (eggDrop (n , k)); } } Python # Function to find minimum number of attempts # needed in order to find the critical floor def eggDrop (n , k): # create a 2D table to store the results dp = [[0 for _ in range (n + 1)] for _ in range (k + 1)] # to count the number of moves cnt = 0 # while the number of floors is less than k while dp [cnt][n] < k : cnt += 1 # for each egg for i in range (1 , n + 1): dp [cnt][i] = 1 + dp [cnt - 1][i - 1] + dp [cnt - 1][i] return cnt if __name__ == "__main__" : n = 2 k = 10 print (eggDrop (n , k)) C# // Function to find minimum number of attempts // needed in order to find the critical floor using System ; class GfG { // Function to find minimum number of attempts // needed in order to find the critical floor static int eggDrop (int n , int k) { // create a 2D table to store the results int [][] dp = new int [k + 1][]; for (int i = 0 ; i <= k ; i ++) { dp [i] = new int [n + 1]; } // to count the number of moves int cnt = 0 ; // while the number of floors is less than k while (dp [cnt][n] < k) { cnt ++ ; // for each egg for (int i = 1 ; i <= n ; i ++) { dp [cnt][i] = 1 + dp [cnt - 1][i - 1] + dp [cnt - 1][i]; } } return cnt ; } static void Main () { int n = 2 , k = 10 ; Console . WriteLine (eggDrop (n , k)); } } JavaScript // Function to find minimum number of attempts // needed in order to find the critical floor function eggDrop (n , k) { // create a 2D table to store the results let dp = new Array (k + 1); for (let i = 0 ; i <= k ; i ++) { dp [i] = new Array (n + 1). fill (0); } // to count the number of moves let cnt = 0 ; // while the number of floors is less than k while (dp [cnt][n] < k) { cnt ++ ; // for each egg for (let i = 1 ; i <= n ; i ++) { dp [cnt][i] = 1 + dp [cnt - 1][i - 1] + dp [cnt - 1][i]; } } return dp [cnt][n] >= k ? cnt : cnt ; } let n = 2 , k = 10 ; console . log (eggDrop (n , k)); Output 4 [Optimized Approach] Using S pace-Optimized Table - O(n * k) Time and O(n) Space In the above approach, to calculate the current row of the dp[][] table, we require only the previous row results. The idea is to create a 1D array dp[] of size n to store the result for each move. To do so, proceed similar to above approach, and for each iteration, set dp[i] = 1 + dp[i] + dp[i-1]. At last return the value stored in dp[n]. Illustration for 2 Eggs and 10 Floors (n = 2, k = 10) C++ #include <bits/stdc++.h> using namespace std ; // Function to find minimum number of attempts // needed in order to find the critical floor int eggDrop (int n , int k) { // create an array to store the results vector < int > dp (n + 1 , 0); // to count the number of moves int cnt = 0 ; // while the number of floors is less than k while (dp [n] < k) { cnt ++ ; // for each egg for (int i = n ; i > 0 ; i --) { dp [i] += 1 + dp [i - 1]; } } return cnt ; } int main () { int n = 2 , k = 10 ; cout << eggDrop (n , k); return 0 ; } Java // Function to find minimum number of attempts // needed in order to find the critical floor import java.util.* ; class GfG { // Function to find minimum number of attempts // needed in order to find the critical floor static int eggDrop (int n , int k) { // create an array to store the results int [] dp = new int [n + 1] ; // to count the number of moves int cnt = 0 ; // while the number of floors is less than k while (dp [n] < k) { cnt ++ ; // for each egg for (int i = n ; i > 0 ; i --) { dp [i] += 1 + dp [i - 1] ; } } return cnt ; } public static void main (String [] args) { int n = 2 , k = 10 ; System . out . println (eggDrop (n , k)); } } Python # Function to find minimum number of attempts # needed in order to find the critical floor def eggDrop (n , k): # create an array to store the results dp = [0] * (n + 1) # to count the number of moves cnt = 0 # while the number of floors is less than k while dp [n] < k : cnt += 1 # for each egg for i in range (n , 0 , - 1): dp [i] += 1 + dp [i - 1] return cnt if __name__ == "__main__" : n = 2 k = 10 print (eggDrop (n , k)) C# // Function to find minimum number of attempts // needed in order to find the critical floor using System ; class GfG { // Function to find minimum number of attempts // needed in order to find the critical floor static int eggDrop (int n , int k) { // create an array to store the results int [] dp = new int [n + 1]; // to count the number of moves int cnt = 0 ; // while the number of floors is less than k while (dp [n] < k) { cnt ++ ; // for each egg for (int i = n ; i > 0 ; i --) { dp [i] += 1 + dp [i - 1]; } } return cnt ; } static void Main () { int n = 2 , k = 10 ; Console . WriteLine (eggDrop (n , k)); } } JavaScript // Function to find minimum number of attempts // needed in order to find the critical floor function eggDrop (n , k) { // create an array to store the results let dp = new Array (n + 1). fill (0); // to count the number of moves let cnt = 0 ; // while the number of floors is less than k while (dp [n] < k) { cnt ++ ; // for each egg for (let i = n ; i > 0 ; i --) { dp [i] += 1 + dp [i - 1]; } } return cnt ; } let n = 2 , k = 10 ; console . log (eggDrop (n , k)); Output 4 Related Articles: Eggs dropping puzzle (Binomial Coefficient and Binary Search Solution) 2 Eggs and 100 Floor Puzzle Comment Article Tags: Article Tags: Dynamic Programming DSA Egg-Dropping