Egg Dropping Puzzle

Monday, November 29, 2021 3:34 PM

Egg Dropping Puzzle | DP-11

- Difficulty Level : Hard
- Last Updated : 23 Nov, 2021

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 the total number of trials are minimized.

Source: *Wiki for Dynamic Programming*

Recommended: Please solve it on “***PRACTICE*** ” first, before moving on to the solution.

In case you wish to attend **live classes** with experts, please refer **DSA Live Classes for Working Professionals** and **Competitive Programming Live for Students**.

Method 1: Recursion.

In this post, we will discuss a solution to a general problem with 'n' eggs and 'k' floors. The solution 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. The floor which gives the minimum value in the worst case is going to be part of the solution.

In the following solutions, we return the minimum number of trials in the worst case; these solutions can be easily modified to print floor numbers of every trial also.

Meaning of a worst-case scenario: Worst case scenario gives the user the surety of the threshold floor. For example- If we have '1' egg and 'k' floors, we will start dropping the egg from the first floor till the egg breaks suppose on the 'kth' floor so the number of tries to give us surety is 'k'.

1) Optimal Substructure:

When we drop an egg from a floor x, there can be two cases (1) The egg breaks (2) The egg doesn't break.

1. If the egg breaks after dropping from 'xth' floor, then we only need to check for floors lower than 'x' with remaining eggs as some floor should exist lower than 'x' in which egg would not break; so the problem reduces to x-1 floors and n-1 eggs.
2. If the egg doesn't break after dropping from the 'xth' floor, then we only need to check for floors higher than 'x'; so the problem reduces to 'k-x' floors and n eggs.
   Since we need to minimize the number of trials in *worst* case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials.

k ==> Number of floors

n ==> Number of Eggs

eggDrop(n, k) ==> Minimum number of trials needed to find the critical

floor in worst case.

eggDrop(n, k) = 1 + min{max(eggDrop(n – 1, x – 1), eggDrop(n, k – x)), where x is in {1, 2, …, k}}

Concept of worst case:

For example :

Let there be '2' eggs and '2' floors then-:

If we try throwing from '1st' floor:

Number of tries in worst case= 1+max(0, 1)

0=>If the egg breaks from first floor then it is threshold floor (best case possibility).

1=>If the egg does not break from first floor we will now have '2' eggs and 1 floor to test which will give answer as

'1'.(worst case possibility)

We take the worst case possibility in account, so 1+max(0, 1)=2

If we try throwing from '2nd' floor:

Number of tries in worst case= 1+max(1, 0)

1=>If the egg breaks from second floor then we will have 1 egg and 1 floor to find threshold floor.(Worst Case)

0=>If egg does not break from second floor then it is threshold floor.(Best Case)

We take worst case possibility for surety, so 1+max(1, 0)=2.

The final answer is min(1st, 2nd, 3rd….., kth floor)

So answer here is '2'.

Below is the implementation of the above approach:

- C++
- C
- Java
- Python 3
- C#
- Javascript

```
public class GFG {

    /* Function to get minimum number of
    trials needed in worst case with n
    eggs and k floors */
    static int eggDrop(int n, int k)
    {
        // If there are no floors, then
        // no trials needed. OR if there
        // is one floor, one trial needed.
        if (k == 1 || k == 0)
            return k;

        // We need k trials for one egg
        // and k floors
        if (n == 1)
            return k;
        int min = Integer.MAX_VALUE;
        int x, res;

        // Consider all droppings from
        // 1st floor to kth floor and
        // return the minimum of these
        // values plus 1.
        for (x = 1; x <= k; x++) {
            res = Math.max(eggDrop(n - 1, x - 1),
                           eggDrop(n, k - x));
            if (res < min)
                min = res;
        }

        return min + 1;
    }

    // Driver code
    public static void main(String args[])
    {
        int n = 2, k = 10;
        System.out.print("Minimum number of "
                         + "trials in worst case with "
                         + n + " eggs and " + k
                         + " floors is " + eggDrop(n, k));
    }
    // This code is contributed by Ryuga.
}
```

Output

```
Minimum number of trials in worst case with 2 eggs and 10 floors is 4
```

Output:

```
Minimum number of trials in worst
case with 2 eggs and 10 floors is 4
```

It should be noted that the above function computes the same subproblems again and again. See the following partial recursion tree, E(2, 2) is being evaluated twice. There will many repeated subproblems when you draw the complete recursion tree even for small values of n and k.

```
                          E(2, 4)
                            |
          --------------------------------------
           |             |            |         |
           |             |            |         |
      x=1/           x=2/      x=3/     x=4/
         /              /          ....      ....
       /              /
 E(1, 0)  E(2, 3)      E(1, 1)  E(2, 2)
           /  /...         /
      x=1/                  .....
```

```
        /
     E(1, 0)  E(2, 2)
             /
             ......
Partial recursion tree for 2 eggs and 4 floors.
```

Complexity Analysis:

- **Time Complexity:** As there is a case of overlapping sub-problems the time complexity is exponential.
- **Auxiliary Space :**O(1). As there was no use of any data structure for storing values.
  Since same subproblems are called again, this problem has Overlapping Subproblems property. So Egg Dropping Puzzle has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array eggFloor[][] in bottom up manner.

Method 2: Dynamic Programming.

In this approach, we work on the same idea as described above **neglecting the case of calculating the answers to sub-problems again and again.**. The approach will be to make a table which will store the results of sub-problems so that to solve a sub-problem, it would only require a look-up from the table which will take **constant time**, which earlier took **exponential time**.

Formally for filling DP[i][j] state where 'i' is the number of eggs and 'j' is the number of floors:

- We have to traverse for each floor 'x' from '1' to 'j' and find minimum of:
  `(1 + max( DP[i-1][j-1], DP[i][j-x] )).`
  This simulation will make things clear:

i => Number of eggs

j => Number of floors

Look up find maximum

Lets fill the table for the following case:

Floors = '4'

Eggs = '2'

1 2 3 4

1 2 3 4 => 1

1 2 2 3 => 2

For ‘egg-1’ each case is the base case so the

number of attempts is equal to floor number.

For ‘egg-2’ it will take ‘1’ attempt for 1st

floor which is base case.

For floor-2 =>

Taking 1st floor 1 + max(0, DP[1][1])

Taking 2nd floor 1 + max(DP[1][1], 0)

DP[2][2] = min(1 + max(0, DP[1][1]), 1 + max(DP[1][1], 0))

For floor-3 =>

Taking 1st floor 1 + max(0, DP[2][2])

Taking 2nd floor 1 + max(DP[1][1], DP[2][1])

Taking 3rd floor 1 + max(0, DP[2][2])

DP[2][3]= min(‘all three floors’) = 2

For floor-4 =>

Taking 1st floor 1 + max(0, DP[2][3])

Taking 2nd floor 1 + max(DP[1][1], DP[2][2])

Taking 3rd floor 1 + max(DP[1][2], DP[2][1])

Taking 4th floor 1 + max(0, DP[2][3])

DP[2][4]= min(‘all four floors’) = 3

- C++
- C
- Java
- Python
- C#
- PHP
- Javascript

```
// A Dynamic Programming based Java
// Program for the Egg Dropping Puzzle
class EggDrop {

    // A utility function to get
    // maximum of two integers
    static int max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    /* Function to get minimum number
 of trials needed in worst
    case with n eggs and k floors */
    static int eggDrop(int n, int k)
    {
        /* A 2D table where entry eggFloor[i][j]
 will represent minimum number of trials
needed for i eggs and j floors. */
        int eggFloor[][] = new int[n + 1][k + 1];
        int res;
        int i, j, x;

        // We need one trial for one floor and
        // 0 trials for 0 floors
        for (i = 1; i <= n; i++) {
            eggFloor[i][1] = 1;
            eggFloor[i][0] = 0;
        }

        // We always need j trials for one egg
        // and j floors.
        for (j = 1; j <= k; j++)
            eggFloor[1][j] = j;

        // Fill rest of the entries in table using
        // optimal substructure property
        for (i = 2; i <= n; i++) {
            for (j = 2; j <= k; j++) {
                eggFloor[i][j] = Integer.MAX_VALUE;
                for (x = 1; x <= j; x++) {
                    res = 1 + max(
                                  eggFloor[i - 1][x - 1],
                                  eggFloor[i][j - x]);
                    if (res < eggFloor[i][j])
                        eggFloor[i][j] = res;
                }
            }
        }

        // eggFloor[n][k] holds the result
        return eggFloor[n][k];
    }

    /* Driver program to test to print printDups*/
    public static void main(String args[])
    {
        int n = 2, k = 10;
        System.out.println("Minimum number of trials in worst"
                           + " case with "
                           + n + "  eggs and "
                           + k + " floors is " + eggDrop(n, k));
    }
}
```

```
/*This code is contributed by Rajat Mishra*/
```
Output

```
Minimum number of trials in worst case with 2 eggs and 36 floors is 8
```
Complexity Analysis:

- **Time Complexity:** O(n*k^2).
  Where 'n' is the number of eggs and 'k' is the number of floors, as we use a nested for loop 'k^2' times for each egg
- **Auxiliary Space:** O(n*k).
  As a 2-D array of size 'n*k' is used for storing elements.
  Method 3: Dynamic Programming using memoization.

- C++
- Java
- Python3
- C#
- Javascript

```
import java.util.Arrays;

class GFG {
    static final int MAX = 1000;

    static int[][] memo = new int[MAX][MAX];

    static int solveEggDrop(int n, int k)
    {

        if (memo[n][k] != -1) {
            return memo[n][k];
        }

        if (k == 1 || k == 0)
            return k;

        if (n == 1)
            return k;

        int min = Integer.MAX_VALUE, x, res;

        for (x = 1; x <= k; x++) {
            res = Math.max(solveEggDrop(n - 1, x - 1),
                           solveEggDrop(n, k - x));
            if (res < min)
                min = res;
        }

        memo[n][k] = min + 1;
        return min + 1;
    }

    public static void main(String[] args)
    {
        for (int i = 0; i < memo.length; i++)
            Arrays.fill(memo[i], -1);
        int n = 2, k = 36;
        System.out.print(solveEggDrop(n, k));
    }
}

// This code IS contributed by umadevi9616
```

Output

```
8
```

As an exercise, you may try modifying the above DP solution to print all intermediate floors (The floors used for minimum trial solution).

More Efficient Solution : Eggs dropping puzzle (Binomial Coefficient and Binary Search Solution)

Egg Dropping Puzzle with 2 Eggs and K Floors

2 Eggs and 100 Floor Puzzle

From <https://www.geeksforgeeks.org/egg-dropping-puzzle-dp-11/?ref=lbp>