Egg Dropping Puzzle | DP-11

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 does 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 are 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 is minimized.

Note: 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 the floor numbers of every trial also.

What is 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 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 floors should exist lower than 'x' in which the 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 the *worst* case, we take a maximum of two cases. We consider the max of the above two cases for every floor and choose the floor which yields the minimum number of trials.
Below is the illustration of the above approach:

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:
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++ program for the above approach


#include <bits/stdc++.h>
using namespace std;


// A utility function to get
// maximum of two integers
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
int eggDrop(int n, int k)
{
    // If there are no floors,
```

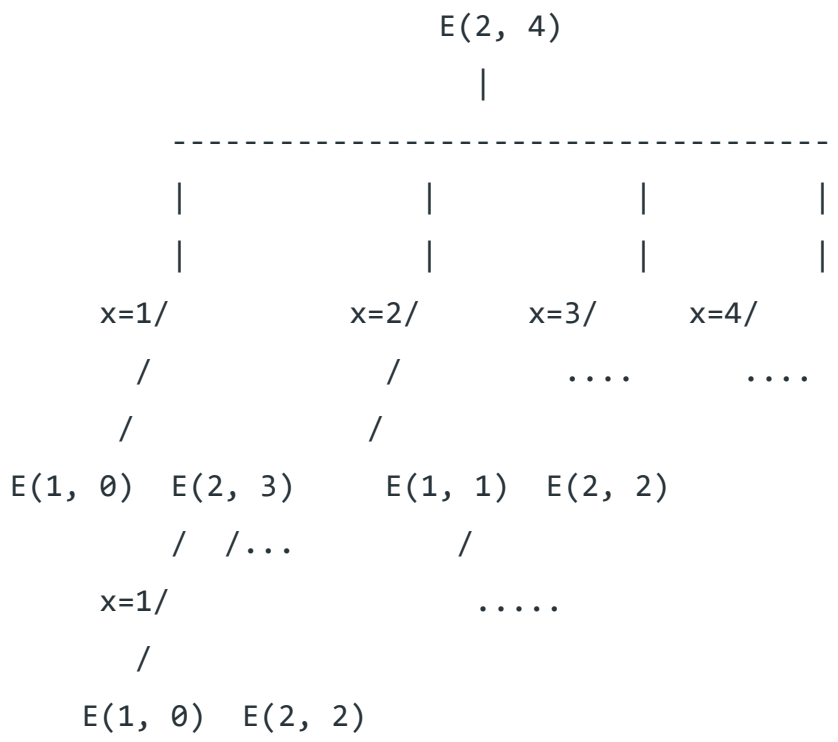
```
    // then no trials needed.
    // OR if there is one floor,
    // one trial needed.
    if (k == 1 || k == 0)
        return k;

    // We need k trials for one
    // egg and k floors
    if (n == 1)
        return k;

    int min = INT_MAX, 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 = max(eggDrop(n - 1, x - 1), eggDrop(n, k - x));
        if (res < min)
            min = res;
    }

    return min + 1;
}

// Driver code
```

```
int main()
{
    int n = 2, k = 10;
    cout << "Minimum number of trials "
            "in worst case with "
         << n << " eggs and " << k << " floors is "
         << eggDrop(n, k) << endl;
    return 0;
}
```

Output

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

Note: 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 be many repeated subproblems when you draw the complete recursion tree even for small values of N and K.



```
          /
          ......
```

Partial recursion tree for 2 eggs and 4 floors.

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.

Egg Dropping Puzzle using Dynamic Programming:
To solve the problem follow the below idea:

Since the same subproblems are called again, this problem has the 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 the same subproblems can be avoided by constructing a temporary array eggFloor[][] in a bottom-up manner.
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 that will store the results of sub-problems so that to solve a sub-problem, 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 a minimum of:

(1 + max(DP[i-1][j-1], DP[i][j-x])).

Below is the illustration of the above approach:

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

Below is the implementation of the above approach:

C++

```
// C++ program for the above approach


#include <bits/stdc++.h>
using namespace std;
```

```
// A utility function to get
// maximum of two integers
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 */
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[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] = INT_MAX;
            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 code
int main()
{
    int n = 2, k = 36;
    cout << "\nMinimum number of trials "
```

```
            "in worst case with "
         << n << " eggs and " << k << " floors is "
         << eggDrop(n, k);
    return 0;
}
```

Output

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

Time Complexity: O(N * K^2). As we use a nested for loop 'k^2' times for each egg

Auxiliary Space: O(N * K). A 2-D array of size 'n*k' is used for storing elements.

Egg Dropping Puzzle using Memoization:

To solve the problem follow the below idea:

We can use a 2D dp table in the first recursive approach to store the results of overlapping subproblems which will help to reduce the time complexity from exponential to quadratic

dp table state –> dp[i][j], where where 'i' is the number of eggs and 'j' is the number of floors:

Follow the below steps to solve the problem:

- Declare a 2-D array memo of size N+1 * K+1 and call the function solveEggDrop(N, K)
- If memo[N][K] is already computed then return memo[n][k]
- If K == 1 or K == 0 then return K (Base Case)
- If N == 1 return K (Base case)
- Create an integer min equal to the integer Maximum and an integer res
- Run a for loop from x equal to 1 till x is less than or equal to K
    - Set res equal to maximum of solveEggDrop(N-1, x-1) or solveEggDrop(N, k-x)
    - If res is less than integer min then set min equal to res

- Set memo[N][K] equal to min + 1
- Return min + 1

Below is the implementation of the above approach:

C++

```
// C++ program for the above approach


#include <bits/stdc++.h>
using namespace std;
#define MAX 1000


vector<vector<int> > memo(MAX, vector<int>(MAX, -1));
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 = INT_MAX, x, res;

    for (x = 1; x <= k; x++) {
        res = max(solveEggDrop(n - 1, x - 1),
```

```
                    solveEggDrop(n, k - x));
            if (res < min)
                min = res;
        }

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

// Driver code
int main()
{

        int n = 2, k = 36;
          cout << "Minimum number of trials "
                "in worst case with "
             << n << " eggs and " << k << " floors is ";
        cout << solveEggDrop(n, k);
        return 0;
}
```

Output

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

Time Complexity: O(n*k*k) where n is number of eggs and k is number of floors.

Auxiliary Space: O(n*k) as 2D memoisation table has been used.

Approach: To solve the problem follow the below idea:

The approach with O(N * K^2) has been discussed before, where dp[N][K] = 1 + max(dp[N – 1][i – 1], dp[N][K – i]) for i in 1...K. You checked all the possibilities in that approach.
Consider the problem in a different way:
dp[m][x] means that, given x eggs and m moves,
what is the maximum number of floors that can be checked

The DP equation is: dp[m][x] = 1 + dp[m – 1][x – 1] + dp[m – 1][x],
which means we take 1 move to a floor.
If egg breaks, then we can check dp[m – 1][x – 1] floors.
If egg doesn't break, then we can check dp[m – 1][x] floors.

Follow the below steps to solve the problem:

- Declare a 2-D array of size K+1 * N+1 and an integer m equal to zero
- While dp[m][n] < k
    - increase 'm' by 1 and run a for loop from x equal to one till x is less than or equal to n
    - Set dp[m][x] equal to 1 + dp[m-1][x-1] + dp[m-1][x]
- Return m

Below is the implementation of the above approach:

C++

```
// C++ program to find minimum number of trials in worst
// case.
#include <bits/stdc++.h>
using namespace std;


int minTrials(int n, int k)
{
    // Initialize 2D of size (k+1) * (n+1).
    vector<vector<int> > dp(k + 1, vector<int>(n + 1, 0));
    int m = 0; // Number of moves
    while (dp[m][n] < k) {
```

```
        m++;
        for (int x = 1; x <= n; x++) {
            dp[m][x] = 1 + dp[m - 1][x - 1] + dp[m - 1][x];
        }
    }
    return m;
}

/* Driver code*/
int main()
{
      int n = 2, k = 36;
      cout<<"Minimum number of trials "
            "in worst case with "
         << n << " eggs and " << k << " floors is ";
    cout << minTrials(2, 36);
    return 0;
}
```

Output

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

Time Complexity: O(N * K)
Auxiliary Space: O(N * K)

Egg Dropping Puzzle using space-optimized DP:

The fourth approach can be optimized to 1-D DP as for calculating the current row of the DP table, we require only the previous row results and not beyond that.

Follow the below steps to solve the problem:

- Create an array dp of size N+1 and an integer m

- Run a for loop from m equal to zero till dp[n] < k
    - Run a nested for loop from x equal to n till x is greater than zero
    - Set dp[x] equal to 1 + dp[x-1]
- Return m

Below is the implementation of the above approach:

C++

```
// C++ program to find minimum number of trials in worst
// case.
#include <bits/stdc++.h>
using namespace std;


int minTrials(int n, int k)
{
    // Initialize array of size (n+1) and m as moves.
    int dp[n + 1] = { 0 }, m;
    for (m = 0; dp[n] < k; m++) {
        for (int x = n; x > 0; x--) {
            dp[x] += 1 + dp[x - 1];
        }
    }
    return m;
}


/* Driver code*/
int main()
{
  int n = 2, k = 36;
  cout<<"Minimum number of trials "
```

```
            "in worst case with "
         << n << " eggs and " << k << " floors is ";
    cout << minTrials(2, 36);
    return 0;
}
```

Output

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

Time Complexity: O(N * log K)
Auxiliary Space: O(N)