Daily Coding Problem #230

Problem

This problem was asked by Goldman Sachs.

You are given `N` identical eggs and access to a building with `k` floors. Your task is to find the lowest floor that will cause an egg to break, if dropped from that floor. Once an egg breaks, it cannot be dropped again. If an egg breaks when dropped from the x^{th} floor, you can assume it will also break when dropped from any floor greater than x.

Write an algorithm that finds the minimum number of trial drops it will take, in the worst case, to identify this floor.

For example, if `N = 1` and `k = 5`, we will need to try dropping the egg at every floor, beginning with the first, until we reach the fifth floor, so our solution will be 5.

Solution

As mentioned above, if we have only one egg, we cannot afford to risk breaking it, since we would have no eggs left with which to experiment. In this case, then, we must start with the first floor and drop the egg one floor at a time. So for `N = 1`, the worst case number of trials is always `k`.

On the other hand, suppose we had an infinite number of eggs. Then an optimal strategy would involve a binary search over the floors. We would first drop an egg from floor `50`. If the egg broke, we would try from floor `25`; if not, we would try from floor `75`. Continuing in this way, we can see that the number of trials would be around `log k`.

Note, though, that binary search is not an effective strategy when `N = 2`. If our first egg breaks from floor `50`, we have no alternative but to try dropping our remaining egg from every floor from `1` to `49`, leading to a worst case of `49` trials. With an arbitrary number of eggs, then, the number of trials should be somewhere between `log k` and `k`. How can we find a more precise solution?

It will help to introduce some terminology. Let `T(N, k)` be the minimum number of trials to determine the correct floor, given `N` eggs and `k` floors, and let `x` be a random floor.

Note that if we drop an egg from the xth floor and it breaks, we will have `N - 1` eggs left to examine the `x - 1` floors below. On the other hand, if this egg does not break, the number of eggs remains `N`, and we can be assured that the breaking-point floor is somewhere between `x + 1` and `k`. Since we are interested in the worst case scenario, we can assume the number of trials after dropping from floor `x` will be one more than the greater of `T(N – 1, x – 1)` and `T(N, k – x)`.

Each turn, then, we want to choose a test floor `x` that minimizes the maximum of these two terms. This insight can be translated into a recursive solution, as follows.

```python
def min_drops(eggs, floors):
    if floors == 0:
        return 0
    if eggs == 0:
        return float("inf")
    return 1 + min([max(min_drops(eggs - 1, x - 1), min_drops(eggs, floors - x))
                    for x in range(1, floors + 1)])
```

The base cases above are:

- If the number of floors is zero, no trials are needed to determine a solution
- If the number of eggs is zero, we cannot find a solution, so the number of trials is infinitely high

While this solution works, it is terribly slow, because we are calling our function for the same input many times. Since we can express `T(N, k)` in terms of the overlapping subproblems `T(N - 1, x - 1)` and `T(N, k - x)`, this is a great candidate for dynamic programming.

We can start by creating an `N x k` grid `trials`, such that `trials[i][j]` will give us the minimum number of trials needed to satisfy the problem with `i` eggs and `j` floors. The base cases above will be satisfied by setting the grid initially to `inf`, and ensuring that the first column is all zeroes. Then we can iteratively find the solution to successively larger values of `i` and `j`, finally returning `trials[N][k]`.

```
def min_drops(eggs, floors):
    trials = [[float("inf") for _ in range(floors + 1)] for _ in range(eggs +
1)]

    for i in range(eggs + 1):
        trials[i][0] = 0

    for i in range(1, eggs + 1):
        for j in range(1, floors + 1):
            trials[i][j] = 1 + min([max(trials[i - 1][x - 1], trials[i][j - x])
                                    for x in range(1, j + 1)])

    return trials[eggs][floors]
```

Each square in the grid will take `O(k)` time to compute, so the overall complexity of this algorithm is `O(N * k`^{2}`)`.