887. Super Egg Drop

Hard · Math · Binary Search · Dynamic Programming · Leetcode Link

Problem Description

The problem presents a situation where you have `k` identical eggs and a building with `n` floors. The goal is to find the highest floor `f` from which you can drop an egg without it breaking. The catch is that there is an unknown threshold floor where any egg dropped from a floor above it will break, and dropping from `f` or lower will not break the egg. To figure out this threshold `f`, you can perform a series of egg drops starting from any floor. If the egg breaks, you lose it permanently. If it doesn't, you can retrieve it and use it again.

Your task is to determine the minimum number of moves needed to find the threshold floor `f` with absolute certainty, employing the best strategy possible given the number of eggs and floors.

Intuition

Discovering `f` involves a trade-off between minimizing the number of moves and ensuring that the eggs are enough to definitively find the threshold. If you have only one egg, you have to start from the first floor and go up one floor at a time, which would result in the worst case of `n` drops. With more eggs, you can attempt a more efficient binary search-like approach.

The solution is not immediately apparent, as we must strike a balance between the risk of breaking an egg and the information gained from each drop. The intuition behind the optimal strategy is to equalize the risk of moving up or down, which leads to a decision-making process at each step. The risk is defined in terms of how many additional drops we might need.

We want to choose a floor `x` to drop from such that the number of drops needed if the egg breaks is as close as possible to the number of drops needed if it does not break. This minimizes the worst-case scenario after each drop, thereby minimizing the overall number of drops needed to discover `f`.

Dynamic Programming (DP) is used here, where we remember past results and use them to construct solutions to new problems. The `dfs` function calculates the minimum number of moves needed with `i` floors and `j` eggs. By caching these results, we avoid recomputing them, drastically increasing efficiency.

The solution uses binary search inside the `dfs` function to find the critical point `x` which equilibrates the worst-case drops below and above `x`. Then we return the smaller of the two risks (plus one for the current drop) and cache this result to use in future calculations. We repeat the process until we have the number of moves needed for `n` floors and `k` eggs.

Solution Approach

The implementation employs a top-down dynamic programming approach, where the solution to the problem is broken down into subproblems that are solved recursively. The data structure that supports this implementation is a hash table implicitly created through the use of the `cache` decorator, which caches the results of the recursive calls thus avoiding duplicate computation.

Here's the walk-through of the implementation:

1. We define a recursive function `dfs(i, j)` which represents the minimum number of moves required to find out the threshold floor `f` with `i` floors and `j` eggs.
2. The recursive function has two base cases:
   - If there are no floors (`i < 1`), no further moves are needed.
   - If we have only one egg (`j == 1`), our only option is to start from the first floor and go up one at a time which takes `i` moves (worst case).
3. The recursive case uses binary search to find the most efficient floor to drop an egg. It initializes two pointers, `l` and `r`, which represent the range within which to search for this optimal floor.
4. A while loop performs the binary search. The middle floor within the current range is computed as `mid = (l + r + 1) >> 1`. We consider two scenarios:
   - Dropping an egg from floor `mid` and it breaks. We now have a problem of `mid - 1` floors and `j - 1` eggs.
   - Dropping an egg from floor `mid` and it doesn't break. We are left with an `i - mid` floors problem and still `j` eggs.
5. For each mid, we calculate the number of moves needed in both scenarios `(a)` and `(b)` respectively. We aim to minimize the maximum risk, so we adjust the binary search range based on whether `(a)` is less than or equal to `(b)` or not.
6. Once the optimal floor to check is found (at the convergence of `l` and `r`), the function returns the maximum of the two calculated moves plus one for the current move, since whether the egg breaks or not, we've used a move.
7. The `@cache` decorator ensures that the results for each pair of `i` and `j` are saved and thus not recalculated multiple times.
8. Finally, we invoke the `dfs(n, k)` function to get the minimum number of moves required with `n` floors and `k` eggs.

The recursive function employs a depth-first search (hence the naming `dfs`) traversing through various scenarios using the given eggs and floors. It cleverly balances the exploration via binary search, which is a significant optimization over a naive approach.

This algorithm is a blend of binary search, for efficiently narrowing down the floors to check, and dynamic programming, which ensures that intermediate solutions are stored for reuse to minimize the total number of calculations needed.

Example Walkthrough

Let's consider a small example where we have `2` identical eggs `(k = 2)` and a `6` floor building `(n = 6)`. We want to find the highest floor from which we can drop an egg without it breaking, using the least number of drops possible.

1. We start by calling the recursive function `dfs(6, 2)` aiming to find the threshold floor `f` in the most efficient way.
2. Since we have more than one egg, we can optimize our search and don't need to check each floor sequentially. We will use a binary search-like approach to decide the floor to test from. We set `l = 1` and `r = 6` as our initial range.
3. During our first iteration in the binary search, we calculate the middle floor, which is `mid = (1 + 6 + 1) >> 1`, which simplifies to `mid = 4`.
4. We now have two scenarios to consider:
   - The egg breaks at `mid` (`floor 4`), which means `f < 4`. We are then left with `3` floors to check and `1` less egg (`dfs(3, 1)`).
   - The egg doesn't break at `mid` (`floor 4`), indicating `f >= 4`. We now have `2` floors left to check with the same `2` eggs (`dfs(2, 2)`).
5. Let's say we start by assuming the egg breaks at `floor 4`. The worst-case scenario will require us to check `3` more floors sequentially `(3 moves)` with our last remaining egg, as we place `l = 1` and `r = 3`.
6. On the other hand, if the egg doesn't break at `floor 4`, we test `floor 5` next (increment `l` to `5`) because if `floor 4` is safe, we don't need to check `floor 4` again.
7. If the egg breaks at `floor 5`, that means `f` is `floor 4`. If it doesn't, we only have `floor 6` left to test. Either way, we are making only `1` additional move for this scenario.
8. We compare the number of moves in both the scenarios, `3` if it breaks and `1` if it doesn't break. Since we want to minimize the worst-case number of moves, we adjust our binary search to check `floor 3` next.
9. If the egg breaks at `floor 3`, we now need to check from `floor 1` up, costing us `2` moves (`2` sequential floors with the last egg). If it doesn't break, we check `floor 4` next, costing us a total of `2` moves (the previously checked `floor 3` plus testing `floor 4` next).
10. The middle ground between `2` and `2` moves is consistent, so we continue this process, balancing the risk until we find `f`.
11. Each result found on each step is cached by our `dfs` function thanks to the `@cache` decorator, avoiding unnecessary recalculations.
12. Eventually, `dfs(6, 2)` returns `3` as the minimum number of moves required to find `f` with certainty.

By employing both dynamic programming and binary search, this highly optimized strategy efficiently reduces the number of moves needed to figure out the threshold floor `f` for any given `k` eggs and `n` floors.

Python Solution

```python
from functools import lru_cache

class Solution:
    def superEggDrop(self, eggs: int, floors: int) -> int:
        # Decorator to cache results of the recursive calls to reduce
        # computation by not recalculating the same scenarios
        @lru_cache(maxsize=None)
        def drop_egg(moves: int, remaining_eggs: int) -> int:
            # Base case: no moves needed if no floors
            if moves == 0:
                return 0
            # Base case: if only one egg left, we need to check each floor
            if remaining_eggs == 1:
                return moves

            # Initialize binary search bounds
            low, high = 1, moves
            # Perform binary search to find the minimum number of moves
            while low < high:
                mid = (low + high) // 2
                # Recursively check dropping egg from the mid floor
                # breaks: we have mid - 1 floors left, and one less egg
                breaks = drop_egg(mid - 1, remaining_eggs - 1)
                # no_break: we have moves - mid floors left with the same number of eggs
                no_break = drop_egg(moves - mid, remaining_eggs)

                # Narrow down the search space based on comparison
                if breaks <= no_break:
                    low = mid
                else:
                    high = mid - 1

            # The result is the max of breaking and not breaking scenarios plus one for the current move
            return max(drop_egg(low - 1, remaining_eggs - 1), drop_egg(moves - low, remaining_eggs)) + 1

        # Call the recursive function starting with all floors and all eggs
        return drop_egg(floors, eggs)

```

Java Solution

```java
class Solution {
    // Memoization cache to hold the results of subproblems
    private int[][] memo;

    // Entry method for calculating the minimum number of moves
    public int superEggDrop(int k, int n) {
        memo = new int[n + 1][k + 1];
        return findMinMoves(n, k);
    }

    // Helper method using Depth-First Search (DFS) to find the minimum moves
    private int findMinMoves(int floors, int eggs) {
        // The base case: if there are no floors, no moves are required
        if (floors < 1) {
            return 0;
        }
        // If there's only one egg, we need to check each floor starting from the first
        if (eggs == 1) {
            return floors;
        }
        // If we have already computed this state, return the stored result
        if (memo[floors][eggs] != 0) {
            return memo[floors][eggs];
        }

        // Binary search to find the critical floor in optimal manner
        int low = 1, high = floors;
        while (low < high) {
            // Note: Using bitwise right-shift to divide by 2
            int mid = (low + high + 1) >> 1;

            // Recursively find outcomes for dropping an egg from the mid floor
            // a: egg breaks (decrease egg count, lower half)
            // b: egg doesn't break (same egg count, upper half)
            int a = findMinMoves(mid - 1, eggs - 1);
            int b = findMinMoves(floors - mid, eggs);

            // Move our search space according to which scenario is worse
            // as we want to prepare for the worst case
            if (a <= b) {
                low = mid;
            } else {
                high = mid - 1;
            }
        }

        // Combine the worst case and add one for the current move
        memo[floors][eggs] = Math.max(findMinMoves(low - 1, eggs - 1), findMinMoves(floors - low, eggs)) + 1;

        // Return the minimum moves required
        return memo[floors][eggs];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>
#include <cstring>
using namespace std;

class Solution {
public:
    // Function to find the minimum number of attempts needed in the worst case to find the critical floor.
    int superEggDrop(int k, int n) {
        // Initialize a memoization table where the rows represent the number of floors
        // and the columns represent the number of eggs available.
        vector<vector<int>> memo(n + 1, vector<int>(k + 1, 0));

        // Define a recursive lambda function to perform depth-first search
        // 'i' represents floors, and 'j' represents eggs.
        function<int(int, int)> dfs = [&](int floors, int eggs) -> int {
            // If there are no floors, no attempts are needed.
            if (floors < 1) {
                return 0;
            }
            // If there is only one egg, we need 'floors' attempts, as we need to start from the first floor.
            if (eggs == 1) {
                return floors;
            }
            // If we have already computed this state, return the result from the memo table.
            if (memo[floors][eggs]) {
                return memo[floors][eggs];
            }
            // Perform a binary search to find the critical floor in the optimal way
            int low = 1, high = floors;
            while (low < high) {
                int mid = (low + high + 1) >> 1;
                int breakCount = dfs(mid - 1, eggs - 1); // Egg breaks
                int notBreakCount = dfs(floors - mid, eggs); // Egg doesn't break
                // We want to balance the worst case of both scenarios (egg breaking and not breaking)
                if (breakCount <= notBreakCount) {
                    low = mid;
                } else {
                    high = mid - 1;
                }
            }
            // After binary search, store the result in the memo table.
            memo[floors][eggs] = max(dfs(low - 1, eggs - 1), dfs(floors - low, eggs)) + 1;
            return memo[floors][eggs];
        };

        // Call our recursive function starting with 'n' floors and 'k' eggs.
        return dfs(n, k);
    }
};

```

Typescript Solution

```typescript
// Function to calculate the minimum number of attempts needed to find the critical floor
// from which an egg will break, given k eggs and n floors.
function superEggDrop(k: number, n: number): number {
    // Memoization table where `dp[n][k]` will represent the minimum number of attempts
    // needed to find the critical floor with `n` floors and `k` eggs.
    const dp: number[][] = new Array(n + 1).fill(0).map(() => new Array(k + 1).fill(0));

    // Helper function using Depth-First Search approach to find the minimum number of attempts needed.
    function dfs(floors: number, eggs: number): number {
        // Base case: no floors require 0 attempts, and 1 floor requires 1 attempt.
        if (floors < 1) {
            return 0;
        }
        // If there's only one egg, we need a number of attempts equal to the number of floors.
        if (eggs === 1) {
            return floors;
        }
        // If result was already calculated, return the stored value from memoization table.
        if (dp[floors][eggs]) {
            return dp[floors][eggs];
        }

        let low = 1;
        let high = floors;
        // Use binary search to minimize the worst-case number of attempts.
        while (low < high) {
            const mid = Math.floor((low + high + 1) / 2);
            const attemptsIfEggBreaks = dfs(mid - 1, eggs - 1); // Egg breaks, check lower half.
            const attemptsIfEggDoesNotBreak = dfs(floors - mid, eggs); // Egg doesn't break, check upper half.

            // We want to balance the number of attempts between the egg breaking and not breaking cases,
            // to ensure the number of attempts is the minimum worst-case scenario.
            if (attemptsIfEggBreaks <= attemptsIfEggDoesNotBreak) {
                low = mid;
            } else {
                high = mid - 1;
            }
        }

        // Store the result in the memoization table and return.
        // Add one to include the current attempt.
        dp[floors][eggs] = Math.max(dfs(low - 1, eggs - 1), dfs(floors - low, eggs)) + 1;
        return dp[floors][eggs];
    }

    // Call the DFS helper function starting with the given number of floors and eggs.
    return dfs(n, k);
}

```

Time and Space Complexity

The provided code defines a `superEggDrop` method that attempts to find the minimum number of attempts required to find the critical floor in a building with `n` floors using `k` eggs, where the critical floor is defined as the lowest floor from which an egg dropped will break.

The time complexity and space complexity analysis for this code is as follows:

Time complexity

The time complexity of the provided code is `O(kn log n)`. This is because we have a memoized depth-first search (DFS) with `dfs(n, k)` calls. For each state `(i, j)` corresponding to `i` floors and `j` eggs, we perform a binary search to find the minimum attempts which run in `O(log i)`. Since `i` can go up to `n` and we need to compute this for every egg from 1 to `k`, the time complexity is the product of these values.

To express this in a formula, we have:

- `T(k, n)` being the time complexity for `k` eggs and `n` floors,
- `T(k, n) = k * O(n log n)` since we run the binary search (`O(log n)`) for each floor up to `n` for each egg.

Thus, the time complexity is `O(kn log n)`.

Space complexity

The space complexity of the code is `O(kn)` due to the memoization that stores the results of every subproblem `(i, j)`. There are `n` possible floors and `k` possible eggs, hence the space needed to store the results for all subproblems is proportional to the product of these two.

The formula for space complexity is:

- `S(k, n) = k * n`, where `S` denotes space complexity.

Therefore, the space complexity is `O(kn)`.