[2eggs and N floor]

my notes:

"Got 2 eggs. 100 floors. Let's minimize the drama. 🥚🥚 We need a strategy to test the highest drop without cracking under pressure. Here's how you crack it… (pun intended) #MathHacks #AI"

"First egg, go high or go home! Start at floor 14, then 13, 12, and keep reducing until it breaks. 🎢 Why? So that worst-case scenario still gives you a sweet low drop count. 🎯 #Optimized #Algorithm"

Python Code Solution:

python

Copier le code

```
def find_floor(eggs, floors):
    # Start from the floor and decrease the steps after each drop
    step = 0
    while (step * (step + 1)) // 2 < floors:
        step += 1

    first_egg_floor = 0
    drops = 0
    # Drop first egg and find the critical floor
    while first_egg_floor + step <= floors:
        drops += 1
```

```
        print(f"Dropped egg from floor: {first_egg_floor + step}")
        first_egg_floor += step
        step -= 1

    # Now use second egg to find the exact floor between last safe and current floor
    for floor in range(first_egg_floor + 1, first_egg_floor + step + 1):
        drops += 1
        print(f"Second egg test from floor: {floor}")

    return drops

# Example with 2 eggs and 100 floors
total_drops = find_floor(2, 100)
print(f"Minimum number of drops: {total_drops}")
```

"Just cracked the code on 2 eggs and 100 floors. 🥚 + 🏠 = efficient AI magic. In this case, worst-case is just 14 drops! Try it out and drop me a like. 🚀 #TechWins #AIEngineered"

1884. Egg Drop With 2 Eggs and N Floors

Medium

TopicsCompanies

Hint

You are given **two identical** eggs and you have access to a building with n floors labeled from 1 to n.

You know that there exists a floor f where 0 <= f <= n such that any egg dropped at a floor **higher** than f will **break**, and any egg dropped **at or below** floor f will **not break**.

In each move, you may take an **unbroken** egg and drop it from any floor x (where 1 <= x <= n). If the egg breaks, you can no longer use it. However, if the egg does not break, you may **reuse** it in future moves.

Return *the **minimum number of moves** that you need to determine **with certainty** what the value of* f is.

Example 1:

Input: n = 2
Output: 2
Explanation: We can drop the first egg from floor 1 and the second egg from floor 2.
If the first egg breaks, we know that f = 0.
If the second egg breaks but the first egg didn't, we know that f = 1.
Otherwise, if both eggs survive, we know that f = 2.

Example 2:

Input: n = 100
Output: 14
Explanation: One optimal strategy is:
- Drop the 1st egg at floor 9. If it breaks, we know f is between 0 and 8. Drop the 2nd egg starting from floor 1 and going up one at a time to find f within 8 more drops. Total drops is 1 + 8 = 9.
- If the 1st egg does not break, drop the 1st egg again at floor 22. If it breaks, we know f is between 9 and 21. Drop the 2nd egg starting from floor 10 and going up one at a time to find f within 12 more drops. Total drops is 2 + 12 = 14.
- If the 1st egg does not break again, follow a similar process dropping the 1st egg from floors 34, 45, 55, 64, 72, 79, 85, 90, 94, 97, 99, and 100.
Regardless of the outcome, it takes at most 14 drops to determine f.

Constraints:

- 1 <= n <= 1000

This is a classic dynamic programming problem known as the **Egg Drop Problem**. The goal is to find the minimum number of moves to determine the critical floor **f** in a building with **n** floors, using **2 eggs**.

Approach:

The strategy is to minimize the number of moves by using the first egg to reduce the range of possible floors as much as possible, and then use the second egg to narrow it down linearly within the smaller range.

Dynamic Programming Solution:

We can use a dynamic programming (DP) approach to solve this. Let's define:

- `dp[k][m]` as the minimum number of floors we can check with **k** eggs and **m** moves.

The goal is to compute `dp[2][m]` (for 2 eggs and `m` moves) and find the minimum `m` such that `dp[2][m] >= n` (i.e., the number of floors we need to check).

Transition Formula:

- If we drop an egg from a floor, we have two possibilities:
  1. **The egg breaks:** This reduces the problem to `k-1` eggs and `m-1` moves on the floors below.
  2. **The egg doesn't break:** This reduces the problem to `k` eggs and `m-1` moves on the floors above.

Thus, the recursive relation is:

```
css
Copier le code
dp[k][m] = 1 + min(max(dp[k-1][m-1], dp[k][m-1])) for all m
```

Where the term `1 + max(dp[k-1][m-1], dp[k][m-1])` comes from the fact that we drop one egg, and we then reduce the problem either to a smaller number of floors (if it breaks) or a smaller number of moves (if it doesn't break).

Code Implementation:

```
python
Copier le code
def superEggDrop(eggs: int, n: int) -> int:
    # dp[k][m] will be the max number of floors we can check with k eggs
and m moves
    dp = [[0] * (n + 1) for _ in range(eggs + 1)]

    # Fill the DP table for one egg (with k = 1)
```

```
    for m in range(1, n + 1):
        dp[1][m] = m

    # Fill the DP table for all eggs and moves
    for k in range(2, eggs + 1):
        for m in range(1, n + 1):
            dp[k][m] = dp[k-1][m-1] + dp[k][m-1] + 1

    # Find the minimum m such that dp[eggs][m] >= n
    for m in range(1, n + 1):
        if dp[eggs][m] >= n:
            return m

# Example cases
print(superEggDrop(2, 2))  # Output: 2
print(superEggDrop(2, 100))  # Output: 14
```

Explanation:

1. **DP Table Setup:**
   - We initialize a table `dp` where `dp[k][m]` is the maximum number of floors we can test with **`k`** eggs and **`m`** moves.
   - For **`1 egg`**, we need to check all floors linearly, so `dp[1][m] = m` for all `m`.
2. **Fill DP Table:**
   - We compute `dp[k][m]` by considering all possible moves where you drop the egg from a floor and calculate the maximum number of floors you can check after that drop.
3. **Finding the Result:**
   - We find the smallest number of moves `m` such that the number of floors we can test with 2 eggs and `m` moves is at least `n`.

Complexity:

- **Time Complexity:** O(n^2), where `n` is the number of floors. The DP table has size `(eggs+1) x (n+1)` and we fill each cell with O(1) operations.
- **Space Complexity:** O(n^2) due to the space required to store the DP table.

Example Walkthrough:

For `n = 100` floors with 2 eggs, the minimum number of drops required is **`14`**. The strategy would be to drop the first egg from floor 9, then 22, then 34, etc., progressively reducing the search space for the second egg. This ensures that no matter where the egg breaks, the number of drops is minimized.