458. Poor Pigs

Hard Math Dynamic Programming Combinatorics

Problem Description

In this problem, we have a certain number of buckets of liquid, one of which is poisonous. The objective is to identify the poisonous bucket using a given number of pigs within a constrained amount of time. The process involves feeding some of the liquid to the pigs and observing whether they die within a fixed time frame known as `minutesToDie`. This test can be repeated several times within the total time we have, called `minutesToTest`. Each test phase must last for `minutesToDie`, and no pig can be fed more than once during each phase.

The challenge is to determine the minimum number of pigs necessary to ensure that we can correctly identify the single poisonous bucket within the time constraints. The problem can be abstracted into a question about information theory and combinatorial mathematics, where each pig's survival or death provides us with information that helps narrow down the possibilities.

Intuition

The intuition behind the solution is based on the idea that every pig can provide us with information through binary outcomes: either the pig dies (indicating at least one of the buckets it drank from is poisonous) or the pig survives (indicating none of the buckets it drank from contain the poison). Each round of testing (lasting `minutesToDie` minutes) allows us to gather more information.

One way to maximize the information from each pig is to consider a strategy where the outcome of each round allows us to effectively eliminate as many buckets as possible from suspicion. Our goal is to design a testing strategy such that the different combinations of pig deaths map uniquely to each bucket.

We use a base variable to represent the number of different states a pig can be in after a test (dead or alive), plus an additional state for the pigs that are not used in the round. The base is calculated as `minutesToTest // minutesToDie + 1`, meaning how many rounds of tests we can perform plus one more chance (for not testing the pig at all).

We iterate, increasing the number of pigs used (`res` in the solution) and calculating the total number of different configurations we can have with these pigs (`p` in the solution) until the number of configurations is greater than or equal to the number of buckets we have. Since each different configuration can be directly mapped to a bucket, when `p` is equal to or exceeds `buckets`, it means we have enough pigs to uniquely identify the poisonous bucket.

Solution Approach

The implementation of the solution employs a straightforward mathematical approach without the need for complex data structures or algorithms.

In the given solution, we calculate the `base`, which is the number of rounds we can perform within the `minutesToTest`, plus one additional state for not testing the pig. This is done by dividing `minutesToTest` by `minutesToDie` and adding 1.

The variable `base` essentially represents the number of different states we can observe for a single pig over all the testing rounds, treating each round as a chance for the pig to die if poisoned, or to not participate in that round.

Next, two variables are initialized. The variable `res` represents the minimum number of pigs needed, starting from 0, and the variable `p` represents the total possible configurations, which is initially set to 1 (representing the scenario where no pigs are tested).

We use a while loop to iterate and calculate the number of pigs needed. In each iteration:

- We multiply `p` by the `base` to calculate the number of combinations we can generate with an additional pig. Each additional pig exponentially increases the number of combinations due to the binary outcomes of each test round.
- We increment `res` to acknowledge the addition of another pig.

This loop continues until `p` becomes greater than or equal to `buckets`. When `p` is greater than or equal to `buckets`, it indicates that we can uniquely identify each bucket with the number of pigs we have used, because the configurations of alive/dead pigs after all the test rounds can be mapped to a specific bucket.

The moment we reach or exceed the number of buckets with our combinations, we have enough pigs to determine which bucket is poisonous. Therefore, the solution concludes by returning `res` as the minimum number of pigs needed.

In simpler terms, the code is evaluating the expression `base^x >= buckets` where `x` is the smallest integer satisfying the inequality, which represents the minimum number of pigs needed. The loop continues, increasing `x` by 1 each time (`res` in the code) until the left side of the inequality meets or exceeds the number of buckets.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose that we have 1000 buckets of liquid, one of which is poisonous. We have up to 60 minutes to test, and it takes 15 minutes for a poison effect to show up on the pigs (`minutesToDie = 15`). Thus, there can be a total of `minutesToTest // minutesToDie = 60 // 15 = 4` rounds of testing.

Using the approach described, we calculate the `base` as `minutesToTest // minutesToDie + 1 = 4 + 1 = 5`. The base here represents the number of statuses a pig can be in: it could not drink in round 1 and die in round 2, 3, 4, or it could not drink at all (the +1 is for not testing).

Now we iteratively calculate how many pigs (`res`) are necessary where each additional pig gives us exponentially more information due to the different combinations of alive or dead after each round.

Here's the calculation broken down into steps:

1. Initially, `res = 0` and `p = 1` (no pigs and one configuration).
2. After adding one pig, `p = 5^1 = 5` (one pig can be in 5 different states, representing 5 different outcomes after the full test duration).
3. With two pigs, `p = 5^2 = 25` (each pig can be in 5 different states, so together they offer 25 combinations).
4. For three pigs, `p = 5^3 = 125` (with each additional pig, the number of combinations multiplies by the base).
5. With four pigs, `p = 5^4 = 625`.
6. At five pigs, `p = 5^5 = 3125`.

At this stage, `p` (3125) exceeds the number of buckets (1000), meaning that with 5 pigs, we have enough combinations to uniquely identify the poisonous bucket, since each combination can correlate to a different bucket.

So, the minimum number of pigs required in this scenario is `res = 5`. With these 5 pigs, we can design a testing strategy that uniquely identifies the poisonous bucket within 60 minutes.

Solution Implementation

Python

```python
class Solution:
    def poorPigs(self, buckets: int, minutes_to_die: int, minutes_to_test: int) -> int:
        # The base determines the number of states a single pig can test.
        # It is calculated using the total minutes to test divided by the minutes to die
        # A +1 is added because one state is needed to represent the pig not dying
        base = minutes_to_test // minutes_to_die + 1

        # 'pigs_needed' will be the final number of pigs needed
        # 'current_buckets' starts at 1 to represent the initial state where no pigs are dead
        pigs_needed = 0
        current_buckets = 1

        # While we have not enough states to test all buckets, add another pig and multiply
        # the current possible states (current_buckets) by the base (the states each pig can test)
        while current_buckets < buckets:
            current_buckets *= base
            pigs_needed += 1  # Another pig is added

        # Return the total number of pigs needed to test all the buckets within the given time
        return pigs_needed
```

Java

```java
class Solution {
    // Method to calculate the minimum number of pigs needed to find the poisonous bucket
    // within the given testing time limit.
    public int poorPigs(int buckets, int minutesToDie, int minutesToTest) {
        // Calculate the 'base' which is the number of states a pig can be in. It's a reflection of
        // how many times you can test each pig within the total testing time 'minutesToTest'.
        int base = minutesToTest / minutesToDie + 1;

        // Initialize a counter for the number of pigs needed.
        int numberOfPigs = 0;

        // Loop to calculate the number of pigs needed. The idea is to multiply the base by itself
        // until we reach or exceed the number of buckets.
        for (int currentBuckets = 1; currentBuckets < buckets; currentBuckets *= base) {
            // Each time we are able to cover more buckets with current number of pigs, we increase the pig count.
            numberOfPigs++;
        }

        // Return the number of pigs calculated as the result.
        return numberOfPigs;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the minimum number of pigs needed to find the poisonous bucket within the given time constraints.
    int poorPigs(int buckets, int minutesToDie, int minutesToTest) {
        // Calculate the number of tests possible within the total time, adding one for the case where a pig does not die.
        int trials = minutesToTest / minutesToDie + 1;

        // Initialize the number of pigs needed to 0.
        int numPigs = 0;

        // Use the base (trials) to determine the number of pigs needed. One pig can test 'trials' number of buckets at least.
        // We use a loop to iteratively increase the number of pigs to determine the minimum number of pigs needed.
        // The loop condition keeps going as long as the total number of possible tested buckets with the current number
        // of pigs is less than the total number of buckets.
        for (int testedBuckets = 1; testedBuckets < buckets; testedBuckets *= trials) {
            numPigs++; // Increase the number of pigs, as the current number isn't enough.
        }

        // Once the loop is complete, we have the minimum number of pigs required.
        return numPigs;
    }
};
```

TypeScript

```typescript
// Calculate the minimum number of pigs needed to find the poisonous bucket within the given time constraints.
function poorPigs(buckets: number, minutesToDie: number, minutesToTest: number): number {
    // Calculate the number of tests possible within the total time, adding one for the case where a pig does not die.
    const trials: number = Math.floor(minutesToTest / minutesToDie) + 1;

    // Initialize the number of pigs needed to 0.
    let numPigs: number = 0;

    // Use the base (trials) to determine the number of pigs needed. One pig can test 'trials' number of buckets at least.
    // We use a loop to iteratively increase the number of pigs to determine the minimum number of pigs needed.
    // The loop condition keeps going as long as the total number of possible tested buckets with the current number
    // of pigs is less than the total number of buckets.
    for (let testedBuckets: number = 1; testedBuckets < buckets; testedBuckets *= trials) {
        numPigs++; // Increase the number of pigs, as the current number isn't enough.
    }

    // Once the loop is complete, we have the minimum number of pigs required.
    return numPigs;
}
```

```python
class Solution:
    def poorPigs(self, buckets: int, minutes_to_die: int, minutes_to_test: int) -> int:
        # The base determines the number of states a single pig can test.
        # It is calculated using the total minutes to test divided by the minutes to die
        # A +1 is added because one state is needed to represent the pig not dying
        base = minutes_to_test // minutes_to_die + 1

        # 'pigs_needed' will be the final number of pigs needed
        # 'current_buckets' starts at 1 to represent the initial state where no pigs are dead
        pigs_needed = 0
        current_buckets = 1

        # While we have not enough states to test all buckets, add another pig and multiply
        # the current possible states (current_buckets) by the base (the states each pig can test)
        while current_buckets < buckets:
            current_buckets *= base
            pigs_needed += 1  # Another pig is added

        # Return the total number of pigs needed to test all the buckets within the given time
        return pigs_needed
```

Time and Space Complexity

The given code implements an algorithm to calculate the minimum number of pigs to find the poisonous bucket within the allotted test time.

Time Complexity:

The time complexity of the code is `O(N)`, where `N` is the number of necessary iterations to find the result. However, this does not mean the time complexity is linear with respect to the number of buckets, but rather the necessary powers of the `base` until we exceed the number of buckets.

Specifically, the loop iterates until `p >= buckets`, making the number of iterations equivalent to the smallest integer `res` such that `base^res >= buckets`. This is fundamentally a logarithmic operation, `res = ceil(log(buckets) / log(base))`.

Since the loop operates in logarithmic time with respect to the number of buckets, and within the loop, we are only performing constant time operations (multiplication and incrementation), the overall time complexity is `O(log(buckets))`.

Space Complexity:

The space complexity of the code is `O(1)`. This is because the algorithm uses a fixed amount of space: variables `base`, `res`, and `p` are the only variables used, and their space requirement does not scale with the input size. There are no data structures that grow in size relative to the input. Thus, the amount of memory used does not change with the number of buckets.