<https://leetcode.com/problems/poor-pigs/description/>

There are buckets buckets of liquid, where **exactly one** of the buckets is poisonous. To figure out which one is poisonous, you feed some number of (poor) pigs the liquid to see whether they will die or not. Unfortunately, you only have minutesToTest minutes to determine which bucket is poisonous.

You can feed the pigs according to these steps:

Choose some live pigs to feed.

For each pig, choose which buckets to feed it. The pig will consume all the chosen buckets simultaneously and will take no time. Each pig can feed from any number of buckets, and each bucket can be fed from by any number of pigs.

Wait for minutesToDie minutes. You may **not** feed any other pigs during this time.

After minutesToDie minutes have passed, any pigs that have been fed the poisonous bucket will die, and all others will survive.

Repeat this process until you run out of time.

Given buckets, minutesToDie, and minutesToTest, return *the****minimum****number of pigs needed to figure out which bucket is poisonous within the allotted time*.

**Example 1:**

**Input:** buckets = 4, minutesToDie = 15, minutesToTest = 15

**Output:** 2

**Explanation:** We can determine the poisonous bucket as follows:At time 0, feed the first pig buckets 1 and 2, and feed the second pig buckets 2 and 3.

At time 15, there are 4 possible outcomes:

- If only the first pig dies, then bucket 1 must be poisonous.

- If only the second pig dies, then bucket 3 must be poisonous.

- If both pigs die, then bucket 2 must be poisonous.

- If neither pig dies, then bucket 4 must be poisonous.

**Example 2:**

**Input:** buckets = 4, minutesToDie = 15, minutesToTest = 30

**Output:** 2

**Explanation:** We can determine the poisonous bucket as follows:

At time 0, feed the first pig bucket 1, and feed the second pig bucket 2.

At time 15, there are 2 possible outcomes:

- If either pig dies, then the poisonous bucket is the one it was fed.

- If neither pig dies, then feed the first pig bucket 3, and feed the second pig bucket 4.

At time 30, one of the two pigs must die, and the poisonous bucket is the one it was fed.

**Constraints:**

1 <= buckets <= 1000

1 <= minutesToDie <= minutesToTest <= 100

**Attempt 1: 2025-06-27**

**Solution 1: Math (30 min)**

class Solution {

    public int poorPigs(int buckets, int minutesToDie, int minutesToTest) {

        if(buckets <= 1) {

            return 0;

        }

        int pigs = 0;

        int test = minutesToTest / minutesToDie;

        // Each pig can encode this many states

        int base = test + 1;

        // Start with 1 state (0 pigs), means with all pigs how many states encoded

        long totalStates = 1;

        while(totalStates < buckets) {

            // Multiply states by base per new pig

            totalStates \*= base;

            pigs++;

        }

        return pigs;

    }

}

Time Complexity: O(1)

Space Complexity: O(1)

**Refer to Deepseek**

This problem is a classic "poison testing" puzzle that requires mathematical insight rather than a brute-force programming approach. Here's the Java solution with explanation:

**Solution**

class Solution {

public int poorPigs(int buckets, int minutesToDie, int minutesToTest) {

int tests = minutesToTest / minutesToDie;

return (int) Math.ceil(Math.log(buckets) / Math.log(tests + 1));

}

}

**Explanation**

**Problem Understanding**

We need to determine the minimum number of pigs required to identify the poisonous bucket within the given time constraints, where:

Each pig can be used in multiple tests (if it doesn't die)

We have limited testing time (minutesToTest)

Each test takes minutesToDie time to determine if a pig dies

**Key Insight**

This problem can be solved using an information theory approach where each pig represents a dimension in a search space:

**Number of tests possible**: tests = minutesToTest / minutesToDie

**Information per pig**: Each pig can provide (tests + 1) states:

Dies in 1st test

Dies in 2nd test

...

Dies in last test

Survives all tests

**Total information needed**: We need to distinguish between buckets possibilities

**Mathematical Formulation**

The minimum number of pigs needed is the smallest integer x such that:

(tests + 1)^x ≥ buckets

This is equivalent to:

x ≥ log(buckets) / log(tests + 1)

**Solution Breakdown**

Calculate how many tests we can perform: tests = minutesToTest / minutesToDie

The base for our logarithm is (tests + 1) because each pig can provide that many states

Compute the logarithm and round up since we can't have a fraction of a pig

**Example**

For buckets = 1000, minutesToDie = 15, minutesToTest = 60:

tests = 60 / 15 = 4

(4 + 1)^x ≥ 1000 ⇒ 5^x ≥ 1000

x ≥ log(1000)/log(5) ≈ 4.29 ⇒ x = 5 pigs

**Time Complexity**

O(1) - Just a mathematical calculation

**Space Complexity**

O(1) - No additional space used

This solution efficiently calculates the minimum number of pigs needed using logarithmic relationships, demonstrating how mathematical insights can solve seemingly complex problems elegantly.

**But the above solution error out on below input:**

**Input**

**buckets = 125**

**minutesToDie = 1**

**minutesToTest = 4**

**Output = 4**

**Expected = 3**

To solve this problem, we need to determine the minimum number of pigs required to identify the poisonous bucket among a given number of buckets within a limited testing time. The solution involves leveraging mathematical insights to avoid brute-force approaches and efficiently compute the result.

**Approach**

**Problem Analysis**: The problem involves identifying a single poisonous bucket among buckets buckets using pigs. Each test takes minutesToDie minutes, and the total available testing time is minutesToTest minutes. The key insight is that each pig can participate in multiple tests (if it survives), and the number of tests that can be performed is minutesToTest / minutesToDie.

**Mathematical Insight**: Each pig can be used to represent a dimension in a search space. The number of states a pig can represent is (tests + 1), where tests is the number of tests possible. The states include:

The pig dies in one of the tests tests.

The pig survives all tests.

**Formula Derivation**: The problem reduces to finding the smallest integer x (number of pigs) such that (tests + 1)^x >= buckets. This is derived from the fact that each pig contributes logarithmically to the number of distinguishable states.

**Algorithm Selection**: **Instead of using logarithms which might introduce floating-point precision errors, we use an iterative approach. We start with zero pigs and multiply the total distinguishable states by (tests + 1) for each additional pig until the total states meet or exceed the number of buckets.**

**Solution Code**

class Solution {

public int poorPigs(int buckets, int minutesToDie, int minutesToTest) {

if (buckets <= 1) {

return 0;

}

int tests = minutesToTest / minutesToDie;

int base = tests + 1;

int pigs = 0;

long total = 1;

while (total < buckets) {

pigs++;

total \*= base;

}

return pigs;

}

}

**Explanation**

**Initial Check**: If there is only one bucket or none, no pigs are needed (return 0).

**Calculate Tests**: The number of tests possible is computed as minutesToTest / minutesToDie.

**Base Calculation**: The base for the logarithmic calculation is tests + 1, representing the states each pig can indicate.

**Iterative Multiplication**: We initialize total to 1 (representing base^0). For each additional pig, we multiply total by base until total is at least buckets. The number of pigs required is the count of these multiplications.

**Efficiency**: The loop runs in logarithmic time relative to the number of buckets, making it efficient even for the upper constraint (buckets ≤ 1000).

This approach ensures accurate results by avoiding floating-point operations and efficiently computes the minimum pigs needed through iterative multiplication.

**Use the given input example explain what exactly "floating-point precision errors" ?**

**Floating-Point Precision Errors Explained**

The original solution used logarithmic calculations which can lead to floating-point precision errors. Let's break this down with the example input:

**Input**: buckets = 125, minutesToDie = 1, minutesToTest = 4

**Calculations**:

tests = minutesToTest / minutesToDie = 4 / 1 = 4

base = tests + 1 = 5

We need to compute: ceil(log₅(125))(since 5^3 = 125, we know this should be exactly 3)

**What Goes Wrong with Floating-Point?**

**Logarithm Calculation**:

log(125) (natural log) ≈ 4.8283137373023015

log(5) (natural log) ≈ 1.6094379124341003

Ratio = log(125) / log(5) ≈ 3.0000000000000004(slightly over 3 due to floating-point imprecision)

**Ceil Function Behavior**:

Math.ceil(3.0000000000000004) → 4.0
(because the value is *barely* over 3, so ceil rounds up to 4)

**Result**:

Original code returns (int) 4.0 → 4

Expected result is 3

**Why Does This Happen?**

Floating-point numbers have limited precision (about 15-17 decimal digits in Java's double).

Logarithm calculations often produce irrational numbers that cannot be represented exactly in binary floating-point format.

Tiny rounding errors can push the result just over an integer boundary, causing ceil() to round up incorrectly.

**Robust Solution (Avoids Floating-Point)**

class Solution {

public int poorPigs(int buckets, int minutesToDie, int minutesToTest) {

if (buckets <= 1) return 0;

int tests = minutesToTest / minutesToDie;

int base = tests + 1; // Each pig can encode this many states

int pigs = 0;

long totalStates = 1; // Start with 1 state (0 pigs)

while (totalStates < buckets) {

pigs++;

totalStates \*= base; // Multiply states by base per new pig

}

return pigs;

}

}

**How the Robust Solution Works with the Example:**

**Initialize**:

base = 5 (tests + 1)

pigs = 0

totalStates = 1

**Iterate**:

pigs = 1 → totalStates = 1 \* 5 = 5 (5 < 125 → continue)

pigs = 2 → totalStates = 5 \* 5 = 25 (25 < 125 → continue)

pigs = 3 → totalStates = 25 \* 5 = 125 (125 >= 125 → stop)

**Result**: Returns 3 (correct).

**Key Takeaways:**

**Avoid floating-point** for exact integer calculations (especially with logarithms and ceiling).

**Use iterative multiplication** to leverage integer arithmetic, which is precise and avoids rounding errors.

**Complexity**: O(log(buckets)) time, O(1) space (efficient even for max constraints).

This approach guarantees mathematical correctness without being affected by floating-point imprecision.