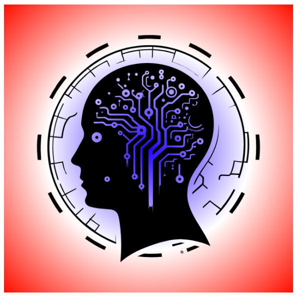
Aryan Burnwal
Student, B.tech
India
`https://www.linkedin.com/in/aryan-burnwal-21a622253`
`https://github.com/Arihant-3`
work.aburnwal@gmail.com



The Thinking Path – Powered by Logic, Framed in LaTeX

A documentation of problem-solving through logic, mathematics, and structured thinking—without leaning on data structures.

By Aryan Burnwal
Timeline: 14 April – 14 May, 2025

Contents

Day 1 . . . **5**
The Problem Statement :- . . . 5
1. My Thought Process . . . 5
2. The Solution . . . 5
3. Actual Intuition . . . 6

Day 2 . . . **6**
The Problem Statement :- . . . 6
1. The Solution . . . 6
2. Actual Intuition . . . 6

Day 3 . . . **7**
The Problem Statement :- . . . 7
1. My Thought Process . . . 7
2. The Solution . . . 8

Day 4 . . . **8**
The Problem Statement :- . . . 8
1. My Thought Process . . . 8
2. The Solution . . . 9
3. Actual Intuition . . . 9

Day 5 **9**
The Problem Statement :- 9
1. The Solution 9
2. Analysis 10

Day 6 **10**
The Problem Statement :- 10
1. The Solution 11
2. Core Intuition 12

Day 7 **12**
The Problem Statement :- 12
1. My Thought Process 13
2. The Solution 13
3. Actual Intuition 13

Day 8 **13**
The Problem Statement :- 13
1. The Solution 14
2. Core Intuition 14

Day 9 **14**
The Problem Statement :- 14
1. My Thought Process 15
2. The Solution 15
3. Actual Intuition 15

Day 10 **16**
The Problem Statement :- 16
1. The Solution 16
2. Step-by-Step Thinking 16

Day 11 **17**
The Problem Statement :- 17
1. My Thought Process 17
2. The Solution 17
3. Actual Intuition 17

Day 12 **18**
The Problem Statement :- 18
1. My Thought Process 18
2. The Solution 19
3. Actual Intuition 19

Day 13 **20**
The Problem Statement :- 20
1. The Solution 20
2. Core Intuition 21

Day 14 **21**
The Problem Statement :- 21

1. My Thought Process . . . 22
2. The Solution . . . 22
3. Actual Intuition . . . 23

Day 15 **23**
The Problem Statement :- . . . 23
1. The Solution . . . 24
2. Core Intuition . . . 25

Day 16 **26**
The Problem Statement :- . . . 26
1. The Solution . . . 26
2. Intuition . . . 27

Day 17 **28**
The Problem Statement :- . . . 28
1. The Solution . . . 28
2. Intuition . . . 29

Day 18 **30**
The Problem Statement :- . . . 30
1. The Solution . . . 30
2. Intuition . . . 31

Day 19 **32**
The Problem Statement :- . . . 32
1. The Solution . . . 32
2. Core Intuition . . . 32

Day 20 **33**
The Problem Statement :- . . . 33
1. The Solution . . . 33
2. Intuition . . . 33

Day 21 **34**
The Problem Statement :- . . . 34
1. The Solution . . . 34
2. Intuition . . . 35

Day 22 **36**
The Problem Statement :- . . . 36
1. The Solution . . . 36
2. Intuition . . . 36

Day 23 **37**
The Problem Statement :- . . . 37
1. The Solution . . . 37
2. Intuition . . . 38

Day 24 **39**
The Problem Statement :- . . . 39

1. The Solution . . . 39
2. Intuition . . . 40

Day 25 **41**
The Problem Statement :- . . . 41
1. My Thought Process . . . 41
2. Actual Logic . . . 42

Day 26 **43**
The Problem Statement :- . . . 43
1. The Solution . . . 43
2. Intuition . . . 44

Day 27 **46**
The Problem Statement :- . . . 46
1. The Solution . . . 46
2. Step-by-Step Analysis . . . 47

Day 28 **49**
The Problem Statement :- . . . 49
1. The Solution . . . 49
2. Analysis . . . 50

Day 29 **51**
The Problem Statement :- . . . 51
1. My Thought Process . . . 51
2. The Solution . . . 51
3. Actual Intuition . . . 51

Day 30 **53**
The Problem Statement :- . . . 53
1. My Thought Process . . . 53
2. Core Idea . . . 54

Day 1 $\longrightarrow$

The Problem Statement :-

Given an integer n, find the number of trailing zeros in $n!$ (i.e., n factorial).

(i) Constraints

– $1 \leq n \leq 10^9$
– You are **not allowed** to compute the actual factorial!

(ii) Hints:

– Where do trailing zeros in numbers come from?
– What factor(s) in multiplication create zeros at the end?
– Can you spot a pattern or rule?

1. My Thought Process

– First, I started thinking about how zeros are formed at the end of a number. I realized that every zero at the end comes from a 10, and 10 is made from multiplying 2 and 5. Since there are usually more 2s than 5s in factorials, the number of 5s basically decides how many zeros we get.

– So, I thought: how often does a 5 appear in the factorial? Every time we have a multiple of 5, we get one 5. That means for every 5 numbers, we gain one trailing zero. But then I noticed something more-numbers like 25 (which is 5^2), 125 (which is 5^3), etc., contribute more than one 5. So we have to count those extra times too.

– That's when I realized the trick is to divide n by 5, then again by 25, then by 125, and so on. Each time we divide and take the floor value, it tells us how many extra 5s are hiding in those powers. This gives the total number of zeros at the end.

2. The Solution

$$\text{Trailing Zeros in } n! = \left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor + \left\lfloor \frac{n}{125} \right\rfloor + \left\lfloor \frac{n}{625} \right\rfloor \cdots$$

3. Actual Intuition

· A trailing zero is created by multiplying a **2 × 5**.
· In $n!$, there are **more 2s than 5s**.
· So, we count how many times **5 is a factor** in all numbers from 1 to n.
But we also need to include **25, 125, 625, ...** because numbers like 25 give **two 5s**, 125 gives **three 5s**, etc.

- -

Day 2 $\longrightarrow$

The Problem Statement :-

Given a non-negative integer n, repeatedly add its digits until the result has only one digit.
Return that final one-digit number.

(i) Constraints:

– $0 \le n \le 2 \times 10^9$
– Try to solve this **without loops or recursion** if possible.

(ii) Hints:

– Is there a **mathematical pattern** in digital root?
– Can you find a shortcut?

1. The Solution

If $n = 0$, then $result = 0$; else, $result = 1 + ((n-1)\, mod\, 9)$

2. Actual Intuition

This is based on a cool math fact called **"mod 9 congruence"** - the sum of digits of any number is **congruent to the number itself modulo 9**.

So instead of looping and summing digits again and again, you can just do: Digital Root of $n = 1 + ((n-1) mod 9)$

Day 3 $\xrightarrow{}$

The Problem Statement :-

A monkey is trying to climb a tree of height H meters.
Each day it climbs up U meters, but every night it slips down D meters.
After how many days will the monkey reach or exceed the top of the tree?

• Inputs:
H : Total height of the tree.
U : Distance the monkey climbs during the day.
D : Distance the monkey slides down at night.

• Output: Return the **number of days** it takes to reach the top.

(i) Constraints:

– The monkey **doesn't slide back** on the **final day** if it reaches or exceeds the height during the day.
– $1 \leq D < U \leq H \leq 10^9$

(ii) Hints:

– Can you find a formula instead of simulating day by day?
– Is there a way to calculate how many **full cycles** the monkey needs before the final climb?

1. My Thought Process

i) Let's say it takes the monkey x days to reach the top of the tree.
ii) On the last day, the monkey climbs up and reaches the top during the day, so it doesn't slide back at night.
iii) This means for the first (x - 1) days, the monkey climbs during the day and slides back at night-so the net progress per day is (U-D) meters.
iv) By the end of the (x-1) days, the monkey must be close enough to the top that one final climb of U meters will get it to or past the height H.
v) So, the total distance that needs to be covered in the first (x-1) days is (H-U).
vi) This gives the equation:

$$(x-1) \times (U-D) = H-U$$

vii) From here, we can solve for x.

2. The Solution

days=$\left\lceil \frac{H-U}{U-D} \right\rceil + 1$

Day 4

$\xrightarrow{}$

The Problem Statement :-

You have n bulbs, all turned **off** initially.
You make n passes over them.
· **On the 1st pass, you toggle every bulb (turn it on).**
· **On the 2nd pass, you toggle every 2nd bulb.**
· **On the 3rd pass, you toggle every 3rd bulb.**
· '...'
· **On the i-th pass, you toggle every i-th bulb.**

After n passes, how many bulbs are **on**?

Hints:

– When does a bulb end up ON after all toggles?
– Try small cases: n = 1 to 10
– Look for a pattern in which positions remain ON.

1. My Thought Process

i) I wasn't sure of the pattern at first, so I decided to try a brute-force approach.
ii) I checked all possible values of n from 1 to 10.
iii) For each value, I manually went through the process to see which switches (or bulbs, etc.) were ON and which were OFF at the end.
iv) I carefully counted how many were ON and how many were OFF for each n.
v) After doing this for a few values, I started noticing a pattern of $\sqrt{5}$ forming based on the positions that remain ON. *vi*) That helped me understand the logic behind the problem better and made it easier to generalize the answer later.

2. The Solution

Number of bulb that remains **ON**=$\lfloor\sqrt{n}\rfloor$

3. Actual Intuition

– A bulb is toggled **once for every divisor** it has.
– Most numbers have **even** number of divisors **(paired up)**.
– But **perfect squares** have **odd number of divisors** - because one of the divisors is repeated (like 4 has divisors 1,2,4 - 2 appears only once).
· So only **bulbs at perfect square positions** (1, 4, 9, 16...) will be ON after all passes.

Day 5

The Problem Statement :-

You are given a list of n integers. Every number appears **exactly twice**, except for **one number that appears only once**.
Find that **unique number** in **O(n)** time and **O(1)** space.

(i) Constraints:

– $1 \leq n \leq 10^6$
– All numbers are non-negative.
– Use no extra space like hash maps or sets.

(ii) Hints:

– Can you use bitwise operators?
– What happens when you XOR a number with itself?

1. The Solution

Use : **XOR**, short for eXclusive OR.
• `^` is same as $\oplus$ for XOR operation

2. Analysis

Let's review the (key properties) of XOR ($\bigoplus$):
1. $a \oplus a = 0 \rightarrow$ A number XORed with itself becomes zero.
2. $a \oplus 0 = a \rightarrow$ A number XORed with zero stays the same.
3. XOR is **commutative** and **associative** :
- $a \oplus b \oplus c = c \oplus b \oplus a$: Order doesn't matter.

Day 6

The Problem Statement :-

You have 8 prison cells, each can be occupied (1) or vacant (0), arranged in a row. Each day, the state of the cells changes according to the following rule:
· **A cell becomes 1 (occupied) if its two adjacent neighbors are both occupied or both vacant.**
· **Otherwise, the cell becomes 0.**
· **First and last cells have no two neighbors, so they always become 0 the next day.**
Given the initial state and a number N, return the state of the prison after N days.

(i) Constraints:

– `initial` is a list of length 8, containing only 0s and 1s.
– $0 \leq n \leq 10^9$ ← yes, **1 billion!**

(ii) Hints:

– What happens if you simulate it naively?
– Do the cell states repeat? If so, can you detect a cycle?
– How many unique configurations are even possible?

1. The Solution

Procedures to Solve It:

1. Simulate day by day, keeping track of seen states.
2. Once you see a repeating state, you've found the cycle length.
3. Now use modulus:

· Instead of simulating all N days,

· Do only N % cycle_length days from the start of the cycle!

Answer

Pseudocode:

```python
def prisonAfterNDays(cells, N):
    seen = {}
    while N > 0:
        state = tuple(cells)
        if state in seen:
            cycle_length = seen[state] - N
            N %= cycle_length
        seen[state] = N

        N -= 1
        next_cells = [0]  # first cell always 0
        for i in range(1, 7):
            next_cells.append(1 if cells[i-1] == cells[i+1] else 0)
        next_cells.append(0)  # last cell always 0
        cells = next_cells
    return cells
```

2. Core Intuition

· To find the state of the prison cells after N days, follow these steps:
i) Each cell updates daily based on its two adjacent neighbors (left and right).
ii) A cell becomes 1 (occupied) if both neighbors are either occupied (1) or both vacant (0).
iii) If the neighbors are different, the cell becomes 0 (vacant).
iv) The first and last cells always become 0, as they do not have two neighbors.
v) Repeat this process for N days, updating all cells simultaneously each day.
vi) Since there are only a limited number of possible states (256), the sequence will eventually enter a cycle (repeat itself).
vii) If a cycle is detected, use it to skip ahead:
Instead of simulating all N days, compute $N \mod \text{cycle\_length}$ and simulate only the remaining steps.
This approach ensures correctness and efficiency, even for large values of N.

Day 7

The Problem Statement :-

You're playing a game with a pile of `n` stones.
You and your friend take turns removing 1, 2, or 3 stones.
The person who removes the **last stone wins**.
You go first. **Can you guarantee a win** if both play optimally?

- Inputs:
`n`: total number of stones.
- Output: Return `True` if you can **guarantee a win**, else `False`.

Hints:

Try small values of `n = 1` to `10`, and look for a pattern:
– Who wins at `n = 1, 2, 3, 4`?
– What do the losing positions have in common?

1. My Thought Process

i) At first, I didn't know the winning strategy, so I decided to use brute force and try out small values of n.
ii) I went from n = 1 to around n = 10 and looked at all possible moves I could make: removing 1, 2, or 3 stones.
iii) For each case, I asked myself:
– "If I take 1 stone now, what will my opponent be left with?"
– "Can they win from there if they play optimally?"
– I repeated this for taking 2 and 3 stones too.
iv) I noticed that if the number of stones was a multiple of 4, I would always end up losing no matter what I played-because whatever I do, my opponent can always bring it back to a multiple of 4.
v) But if it's not a multiple of 4, I can always make a move that gives my opponent a multiple of 4, putting them in the losing position.
vi) So I realized the pattern:
– If n % 4 == 0, I cannot guarantee a win.
– Otherwise, I can guarantee a win.

2. The Solution

return $n \% 4 \neq 0 \rightarrow$ True

3. Actual Intuition

· If `n` is **divisible by 4** (like 4, 8, 12...), you're guaranteed to lose if the opponent plays optimally.
· Any other `n` (1, 2, 3, 5, 6, 7, etc.), you can force your opponent into a **multiple of 4** on their turn.

Day 8 $\longrightarrow$

The Problem Statement :-

Given an integer n, return the number of **1's** in its binary representation (also called **Hamming weight**).

(i) Constraints:

– Use **no string conversion** (bin() or .count() not allowed).
– Solve it using **bitwise logic** only.

(ii) Hints:

– What does $n \mathbin{\&} (n-1)$ do?
– Try running that in a loop - what happens to the number of 1's?

1. The Solution

Code:

Brian Kernighan's Algorithm:

```python
def hammingWeight(n):
    count = 0
    while n:
        n = n & (n - 1)
        count += 1
    return print(count)
```

2. Core Intuition

(*i*) Each time you do $n \mathbin{\&} (n-1)$, it **removes the lowest set bit (1)** from n.
(*ii*) So it runs in O(number of 1s) instead of O(total bits) - much faster for sparse numbers!.

Day 9

The Problem Statement :-

Given a non-negative integer n, repeatedly add its digits until the result has only one digit.
Return that final single-digit number.

(i) Constraints:

– No converting to strings (`str(n)` is a no-go).
– Try to come up with a **constant time** solution - no loop if possible!

(ii) Hints:

– Can you spot a pattern in the result?
– What happens to results for $n = 1 \cdots 20 \cdots 50 \cdots 100 \cdots ?$
– Look into something called the **digital root**.

1. My Thought Process

1. This is the same question as Day 2 but with added constraints.
2. On Day 2, I treated it as a pattern problem.
3. On Day 9, it's an algorithm challenge with no loops and no strings.

2. The Solution

Answer

```python
if n == 0:
    return 0
else:
    return 1 + (n - 1) % 9
```

This is the same question as Day 2. The only difference is:
• On Day 2, It was a pattern problem.
• On this Day 9, It was an algorithm challenge with constraints: no loops, no strings.

3. Actual Intuition

This is based on a cool math fact called "mod 9 congruence" - the sum of digits of any number is congruent to the number itself modulo 9.
So instead of looping and summing digits again and again, you can just do: Digital Root of $n = 1 + ((n - 1) mod 9)$

- -

Day 10

The Problem Statement :-

You're given a positive integer n.
Find the smallest positive integer x such that:
$x \oplus n$ is a perfect square : (where $\bigoplus$ is the bitwise XOR operator)
Return that smallest x.

Hints:

– Can you brute-force from $x = 1$ upwards?
– How can you efficiently check if a number is a perfect square?
– What about the pattern of XOR results?

1. The Solution

Python Code:

```python
import math

def smallest_xor_square(n):
    k = 0
    while True:
        square = k * k
        x = square ^ n
        if x > 0:
            return x
        k += 1
```

2. Step-by-Step Thinking

You want:

$$x \oplus n \equiv \text{some perfect square} = k^2$$

That means:

$$x = k^2 \oplus n$$

So instead of looping through x, you can loop through **perfect squares** k^2, and compute:
Then just check if x is **positive**, and return the first one that works!

Day 11 $\xrightarrow{}$

The Problem Statement :-

Given a positive integer n, determine whether it's possible to divide the set $1, 2, ..., n$ into **two subsets with equal sum**.

Return True if it is possible, otherwise return False.

Hints:

– What's the **sum** of all numbers from 1 to n?
– When is it **even**? Can that guarantee a split??
– Is there a pattern for which n works?

1. My Thought Process

1. I need to check if I can divide the set 1,2,...,n into two subsets with equal sum.
2. First, I calculate the total sum of numbers from 1 to n, which is sum = $\frac{n(n+1)}{2}$
3. If the sum is odd, it's impossible to divide the set into two equal parts, so I return False.
   – I tested values from 1 to 11 and noticed that when the sum is odd, it's not possible to split the set into two equal sums.
4. If the sum is even, I check if it's possible to find a subset with half of that sum using dynamic programming or recursion.

2. The Solution

If $\frac{n(n+1)}{2}$ is even, then return True ; **else** False

3. Actual Intuition

Step-by-Step Reasoning:

· The sum of first n natural numbers is:

$$S = \frac{n(n+1)}{2}$$

· To split this into **two subsets with equal sum**, S must be **even**.
So, all we care about is whether S is divisible by 2 - in other words.

– –

Day 12

The Problem Statement :-

You're given a positive integer n.
You can perform **either** of the following moves:
• **If n is divisible by 3, you can divide it by 3.**
• **If n is divisible by 2, you can divide it by 2.**
• **Subtract 1 from n.**
Goal : Find the **minimum number of steps** to reduce n to 1. Find a simple, efficient way (no need for DP here, think greedy and simple reasoning).

Hints:

– Which move should you prioritize at every step?
– Can you just **greedily** divide by 3 or 2 whenever possible?
– How do you know you're taking the minimal path?

1. My Thought Process

I thought: maybe we can express :

$$n = 2x + 3y + 1$$

This would model the operations as combinations of "divide-by-2" and "divide-by-3" steps, along with an adjustment (like a +1 or similar).
The idea is to represent n in terms of the operations allowed. However, this might not always be straightforward - a purely greedy approach (always choosing one operation when available) may not yield the optimal solution.
So at each step, we may need to carefully consider both options (dividing by 2 or by 3) and sometimes even simulate both paths to see which leads to a better or shorter solution.

2. The Solution

Greedy strategy:

```python
if n % 3 == 0:
    n = n / 3
elif n % 2 == 0:
    n = n / 2
else:
    n = n - 1
count += 1
```

Minimum Steps to One:

```python
def min_steps_to_one(n):
    steps = 0
    while n > 1:
        if n % 3 == 0:
            n //= 3
        elif n % 2 == 0:
            n //= 2
        else:
            n -= 1
        steps += 1
    return steps
```

3. Actual Intuition

You are allowed to:

(i) **Divide by 3** (if divisible)
(ii) **Divide by 2** (if divisible)
(iii) **Subtract 1** (always allowed)

At every step, you want to **minimize** the total number of moves.

Thus, ideal plan:
1. If divisible by 3 $\rightarrow$ **divide by 3** (preferable)
2. Else if divisible by 2 $\rightarrow$ **divide by 2**
3. Else $\rightarrow$ **subtract 1**

Why prefer divide first?
· Dividing shrinks n **much faster** than just subtracting.

• Notes:
– Division shrinks the number much faster than **subtraction**.
– Always check divisibility **by 3 first**, then **by 2**, then **subtract**.

- -

Day 13

The Problem Statement :-

You are given a **positive integer** n.
You must **keep adding the digits** of the number until you get a **single-digit number**, BUT:

- **"If the number is divisible by 9, the final answer should be 9."**
- **"Otherwise, it should be the sum of digits modulo 9."**

Goal : Find the final single-digit number after repeatedly summing digits, using the special rule about multiples of 9.
The question is telling you how you should approach the task, not asking you to solve any particular number directly.

Hints:

This is related to a famous math trick called **"Digital Root"**.

1. The Solution

Answer

```
def strange_sum_of_digits(n):
    if n == 0:
        return 0
    elif n % 9 == 0:
        return 9
    else:
        return n % 9
```

2. Core Intuition

• The sum of the digits of a number gives the same remainder as the number itself when divided by 9.
This is called the **"Digital Root"** property.

– Modulo 9 reduces the number to the same "essence" as adding digits again and again.
– Divisibility by 9 has a special clean behavior (like 81, 99, 729, etc.).
– So using `% 9` and handling the divisible-by-9 case gives you the final single-digit answer quickly.

Day 14

The Problem Statement :-

You have **2 identical eggs** and a building with **100 floors**.
An egg will **break** if dropped from some **floor F or higher**, and will **not break** if dropped from any floor **below F**.
Your goal is to **determine F (the highest safe floor)** in the **fewest drops** possible, in the **worst case**.

Question: What is the **minimum number of drops** you need, in the worst case, to guarantee finding F?

Hints:

– If you drop an egg from floor x:
 · **Breaks:** you have 1 egg left to test all floors **below x** one by one.
 · **Doesn't break**: you still have 2 eggs to test **above x**.
– How should you choose your first drop floor x to **balance** these two scenarios?
– Can you find a sequence of drop floors $x_1, x_2, \cdots$ that minimizes the maximum number of drops?

1. My Thought Process

I first considered jumping in steps of 2, covering all multiples of 2 up to 100 — which gives 50 steps. Next, I considered jumping in steps of 3, covering all multiples of 3 up to 100 — giving 33 steps. However, some numbers like 6 appear in both sequences (i.e., multiples of both 2 and 3), so I subtracted the overlapping numbers. There are 16 such overlaps (multiples of 6), so I calculated the unique numbers as $50+33-16 = 67$. Then, I added the prime numbers less than or equal to 100 that are not multiples of 2 or 3. These are 23 in total, bringing the final count to $67 + 23 = 90$.

2. The Solution

You should drop the first egg at decreasing intervals.
Specifically:
· First drop from floor x ,
· Then $x + (x - 1)$,
· Then $x + (x - 1) + (x - 2)$,
· and so on ,
· so that the total number of drops in worst case is minimized.
This ensures that **no matter when the first egg breaks**, the total number of drops (including the second egg's linear search) will not exceed x.

Finding x:
You want $x + (x - 1) + (x - 2) + \cdots + 1 \geq 100$.
The sum $\frac{x(x+1)}{2} \geq 100$.
Solve:

$$\frac{x(x+1)}{2} \geq 100 \; x(x + 1) \geq 200$$

Approximate:
· Try $x = 14 : 14 \times 15 = 210 \geq 200$
So, $x = 14$.

Minimum number of drops needed = 14

3. Actual Intuition

• **Key Idea:**

· With **one egg**, you could only do **linear search**: drop from floor $1, 2, 3, \cdots$ until it breaks $\rightarrow$ up to 100 drops.
· With **two eggs**, you can be **smarter**:
 · Use the **first egg** to "probe" floors in a smart way.
 · When it breaks, **use the second egg** to linearly search floor by floor below where the first egg broke.

So: The strategy should balance the two:
· If you jump floors too much with the first egg, you may have to linear search a lot with the second.
· If you jump floors too little, you waste moves.
The number of drops **never exceeds 14**, whether the egg breaks early or late.
This is based on a **cumulative search plan**, not prime numbers or divisibility.

• **Simple example:**

Suppose you drop the first egg at floor 14:
· If it breaks, you have at most 1–13 floors below it. You use the second egg linearly to find exactly where.
· If it doesn't break, you go up 13 floors $(14 + 13 = 27)$, and drop again.
· If it doesn't break again, you go up 12 floors $(27 + 12 = 39)$, and so on.
The number of drops **never exceeds 14**, whether the egg breaks early or late.
This is based on a **cumulative search plan**, not prime numbers or divisibility.

Day 15 $\longrightarrow$

The Problem Statement :-

A king has **1000 bottles of wine**, and **one** of them is **poisoned**.
Anyone who drinks the poisoned wine will **die exactly 10 hours later**.
You have **exactly 10 hours** and a supply of **slaves (testers)** to determine which **one** bottle is poisoned.
 • What is the **minimum number of slaves** needed to **guarantee** finding the poisoned bottle in 10 hours?

Rules Recap:

– You can give any number of bottles to each slave.
– All slaves drink **at the same time**.
– Anyone who drinks poisoned wine will die **exactly 10 hours later**.
– You only get **one round of testing**.

Hints:

– Think about **binary representation**.
– What if you number the bottles from 1 to 1000 and make each slave correspond to a **bit**?
– Can the pattern of who lives and dies represent a number?

1. The Solution

You only need **10 slaves** to find the poisoned bottle among 1000.

General Formula:

To uniquely identify n **bottles**, you need:

$$\lceil \log_2(n) \rceil$$

testers.

2. Core Intuition

- Use **Binary encoding**:

· You can represent each bottle number using binary.
· 1000 bottles → you need enough bits to represent the number 1000.
So:

$$2^9 = 512 \textbf{ (not enough)}$$
$$2^{10} = 1028 \textbf{ (covers all 1000 bottles)}$$

Binary Strategy:

– Label each bottle from **1 to 1000**.
– Use **10 slaves** - each represents **one bit position** (from 2^0 to 2^9).
– For each bottle:
  · If a bit is set to 1, that slave drinks from the bottle.
– After 10 hours:
  · The slaves who die correspond to bits set to 1.
  · The resulting binary number tells you exactly which bottle was poisoned. corresponds to the poisoned bottle's number.

- **Step-by-step logic:**

1. Number the 1000 bottles from `0` to `999`.
2. Convert each number to its 10-bit binary form.
3. Assign each of the 10 **slaves** to represent one bit position (e.g., slave 1 = bit 0, slave 2 = bit 1, ..., slave 10 = bit 9).
4. If a bottle has a `1` in a certain bit position, the corresponding slave drinks from that bottle.
5. After 10 hours, each slave will either be alive (`0`) or dead (`1`).
6. The combination of which slaves died gives a **10-bit binary number**, which directly corresponds to the poisoned bottle's number.

- **Example:**

Bottle # 845 → binary is `1101001101`
· Slaves # 1, 3, 4, 6, 7, and 10 drink it.
· If they die, and the others live → read the bits from their deaths = `1101001101` = 845 → **bingo**.

• **Who drinks what?**

Each slave drinks from **all bottles where their assigned bit is 1**.
So:
– A slave may drink from ~ 500 bottles (not all 1000).
– Some bottles make multiple slaves drink (e.g., bottle 511 makes 9 slaves drink).
Yes, this is a **lot of sips per person**, but it's way better than needing 1000 people.

Day 16 $\longrightarrow$

The Problem Statement :-

Define a function $f(n)$ as:
– **If n is even,** $f(n) = \frac{n}{2}$
– **If n is odd,** $f(n) = 3n + 1$
You start with a positive integer n, and repeatedly apply f to the result, until you reach 1.

Question: Which starting number $n \leq 100$ produces the **longest sequence** before reaching 1?

Goal : – Among numbers from 1 to 100, which number produces the **longest** such sequence?
– No need to simulate all 100 manually - try to **guess or reason!**.
– This is a simplified version of the famous **Collatz Conjecture**.

1. The Solution

The starting number $\leq$ 100 that produces the longest sequence is 97, with length 118.

2. Intuition

While the final answer (97) is found using brute-force simulation, there's deeper reasoning behind why this is **necessary** and **non-obvious**:

1. Unpredictable Growth:

– The **Collatz function** mixes two operations: divide by 2 (shrinks) and $3n + 1$ (grows).
– This makes the sequence **non-monotonic** - it doesn't always go down.
– A small number can grow **very large** before reducing to 1.

2. Sequence Length $\neq$ Size:

– Larger starting numbers don't always mean longer sequences.
– For example:
 · 100 → 25 steps
 · 99 → 25 steps
 · 98 → 25 steps
 · 99 → 25 steps
 · 97 → 118 steps **(the longest)**
 · 96 → 12 steps
 · 95 → 105 steps
 · 94 → 105 steps
 · 93 → 17 steps
 · . . .

3. Why Brute Force Is Needed:

– Due to the chaotic and recursive nature of the function, there's **no known formula** to directly calculate sequence length.
– So we use brute force - trying each number and simulating the steps - to find the longest one.

• **So the intuition is:**

Because the **Collatz function** causes numbers to behave chaotically - sometimes rising high before falling - we can't predict the longest sequence analytically. That's why we simulate all values from 1 to 100. Among them, 97 happens to produce the longest journey to 1.

– –

Day 17 $\xrightarrow{}$

The Problem Statement :-

You have a piece of paper that is **0.1 mm thick**.
You start folding it in half, and with **each fold**, the paper's thickness **doubles**.

You fold it once → 0.2 mm
Fold again → 0.4 mm
Then 0.8 mm → 1.6 mm → and so on...

Question: How many times do you have to fold the paper until it becomes **taller than Mount Everest** (assume height of Everest = **8848 meters**)?
Give your **best guess** first — no calculator needed.

Quick Info:

– Starting thickness = 0.1 mm = 0.0001 meters.
– Each fold **doubles** the thickness.

1. The Solution

It takes 27 folds for a **0.1 mm** thick paper to reach or exceed the height of **Mount Everest (8848 meters)**.

2. Intuition

Estimating the Number of Folds to Reach Mount Everest (8848 m) - Intuitive & Calculative Approach:

1. Step-by-Step Reasoning (No Calculator):

– Rearranging the inequality

$$2^n \geq 8848 \times 10^4 = 8.848 \times 10^7$$

Now, mentally estimate when 2^n exceeds 8.848×10^7
– Mental Approximation Using Known Powers of 2
Memorize key benchmarks:
· $2^{3.\cdots} \approx 8.\cdots$
· $2^{10} \approx 10^3 = 1000$
· $2^{20} \approx 10^6 = 1\text{million}$
· $2^{30} \approx 10^9 = 1\text{billion}$
Try:
· $2^{26} = 2^6 \times 2^{20} = 64 \times 10^6 = 64,000,000$
· $2^{27} = 128 \times 2^6 = 128,000,000$
So:

$$2^3 \times 2^{10} \times 2^{10} \times 2^3 = 2^{3+10+10+3} = 2^{26}$$

Then you say, let's call it $2^{3.\cdots} \times 2^{10.\cdots} \times 2^{10.\cdots} \times 2^{3.\cdots} = 2^{(3+10+10+3).\cdots} = 2^{26.\cdots} \approx 2^{27}$ - which is **very reasonable**, because you're compensating for approximation.

2. Calculation (Exact):

– Start from the inequality:

$$10^{-4} \times 2^n \geq 8848 \Rightarrow 2^n \geq 8848 \times 10^4 = 8.848 \times 10^7$$

– Take base-2 logarithm on both sides:

$$n \geq \log_2(8.848 \times 10^7)$$

– Break it down using properties of logs:

$$n \geq \log_2(8.848) + \log_2(10^7) \Rightarrow n \geq \log_2(8.848) + 7\log_2(10)$$

Using known approximations:
· $\log_2(8.848) \approx 3.14$
· $\log_2(10) \approx 3.3219$
So:

$$n \geq 3.14 + 7 \times 3.3219 = 3.14 + 23.2533 = 26.3933$$

Since n must be an integer, $n = 27$ folds are required.

Day 18 $\longrightarrow$

The Problem Statement :-

You are at a party with **100 guests**.
One of them has **poisoned** the wine. You don't know who, but **one** of them is guilty.

Here's what you know:
· The guests were each asked the same yes/no question:
"If you are guilty, you will lie. If you are innocent, you will tell the truth."

· You ask the question:
"If I asked you, 'Are you the poisoner?', would you say yes?"

You get the following answers:
· **99 guests say "Yes"**
· **1 guest says "No"**

Question:
Who is the **poisoner**?
Explain **why**, based on logic.
You don't need to guess - just break it down step by step.

1. The Solution

The guest who said "**No**" is the **poisoner**.
Only a **guilty liar** would say "**No**", to avoid **double-lying**.

2. Intuition

- **Assumptions:**:

· Innocent people always tell the truth.
· The guilty person always lies.

- **Logical Breakdown:**

1. **Innocent Person's Logic:**
· Question: "Are you the poisoner?"
→ Truthful answer: **No**
· Meta-question: "If I asked you, 'Are you the poisoner?', would you say yes?"
→ Since they would answer **No** to the base question, the correct truthful answer to the meta-question is **No**.

2. **Guilty Person's Logic (Liar):**
· Question: "Are you the poisoner?"
→ Lying answer: **No** (denying guilt)
· Meta-question: "Would you say yes?"
→ Truth: they'd say **No**
→ But they lie about it → So they say **Yes**

- **Observed Outcome in the Puzzle:**

· 99 guests say "Yes"
· 1 guest says "No"h̃hh

- **Interpretation:**

– Innocents answer **Yes** (though counterintuitive, this aligns with puzzle logic)
– The guilty liar answers **No**, by double-lying:
  Lies about what they would say → says **No**

- **Conclusion:**

The guest who answered "**No**" is the **poisoner**.

Day 19 $\longrightarrow$

The Problem Statement :-

A clock shows the time as **3:15**.
Now imagine you're looking at the clock in a **mirror** - not a digital one, but an old-school **analog wall clock**.

• **Question: What time will you see in the mirror?**
(Give your answer in the **HH:MM** format, as it would appear on a regular analog clock.)

Hints:

– The mirror is **placed vertically** (as if you're standing in front of the clock).
– Think of **how the positions of the hour and minute hands flip.**

1. The Solution

The mirrored time is:
8:45

2. Core Intuition

Original time: 3:15
– The **hour hand** is slightly ahead of 3
– The **minute hand** is at 3 (since 15 minutes = 3 on the clock)

When mirrored:
– The **minute hand at 3** → reflects to **9**
– The **hour hand slightly ahead of 3** → reflects to slightly before **9**

– –

Day 20
$\xrightarrow{\hspace{2em}}$

The Problem Statement :-

Aliens have a strange way of counting. You're given the following sequence:
1 → 1
2 → 10
3 → 11
4 → 100
5 → 101
6 → 110
7 → 111
8 → 1000
9 → 1001
10 → ?

• **Question:** What number will 10 be converted to in this system?
Also: Can you **figure out the rule** behind the pattern?
(No coding, no math tricks - just pure observation.)

1. The Solution

10 → 1010

2. Intuition

• Observational Pattern:
The more we move forward, the 1 shifts left until there is a 1 after that, and so on - till the number of digits increases.

• Rule Behind the Pattern:
Aliens seem to count using **binary notation**, just like computers do:
– Each number is expressed in base-2 rather than base-10.
– No tricks or alternate logic - just a straightforward binary conversion.

Day 21 $\longrightarrow$

The Problem Statement :-

Three prisoners - A, B, and C - are blindfolded and a hat is placed on each of their heads.
Each hat is either **black or white**.

They're told:

– Hats were chosen from a set of **5 hats: 3 black and 2 white**.
– The guard randomly placed one hat on each prisoner's head (3 total hats used).
– The remaining **2 hats are hidden**.

The prisoners are arranged in a line:

– **C can see B and A**
– **B can see A**
– **A sees no one**

Then, one by one, they're asked:
"Can you tell the color of your own hat?"

– A says: "I don't know."
– B says: "I don't know."
– C says: "I know!"

• **Question: What color is C's hat, and how does C know it?**
Take your time to think it through logically.

1. The Solution

C is wearing a black hat.

2. Intuition

• **Step-by-step deduction:**

i) A says: "I don't know"
– A has no information - can't see anyone → So this is expected

ii) B says: "I don't know"
– B **can see A's hat**
– **Case:**
1. If A had a **white hat**, then B could be wearing a **white hat or black hat** → **uncertain**
2. If A had a **black hat**, then B could be wearing a **white hat or black hat** → **uncertain**
– So B says: "I don't know"

iii) C says: "I know!"
– C sees **both B and A**
– **Case:**
1. If C sees **2 white hats**, then C knows **all white hats are used**, so C must have a **black hat**
2. If C sees anything else (like one or both black), uncertainty remains → C **couldn't be sure**.

⇒ **Implication:** And since A and B both didn't know, the only possibility that lets C confidently say his hat color is:

A and B are both wearing white hats
→ So C must be wearing a black hat

Day 22 $\xrightarrow{\quad}$

The Problem Statement :-

You're given this strange set of "equations":

$1 + 4 = 5$
$2 + 5 = 12$
$3 + 6 = 21$
$8 + 11 = ?$

Clearly, normal addition isn't happening here.

• **Question:** What number should replace the ?
And - more importantly - **what's the pattern**?

1. The Solution

$$\mathbf{8 + 11 = (8 \times 11) + 8 = 88 + 8 = 96}$$

2. Intuition

• **Step-by-step deduction:**

i) 1 + 4 = 5
– Looks like normal addition → nothing seems off
– But let's keep observing

ii) 2 + 5 = 12
– Regular addition gives 7, not 12
– Try combining multiplication: $(2 \times 5) + 2 = 10 + 2 = \mathbf{12}$
→ A new pattern may be forming: $\mathbf{(x \times y) + x}$

iii) 3 + 6 = 21
– Using the same pattern: $(3 \times 6) + 3 = 18 + 3 = \mathbf{21}$
– **Pattern confirmed** in multiple cases

iv) Apply to 8 + 11
– $(8 \times 11) + 8 = 88 + 8 = \mathbf{96}$

⇒ **Implication:**

· The puzzle disguises a formula within fake "addition" equations
· Recognizing the hidden rule requires testing non-standard operations
· Final pattern: $\mathbf{x + y = (x \times y) + x}$

- -

Day 23

$\overrightarrow{}$

The Problem Statement :-

You are on an island that is **perfectly circular** and completely surrounded by water. At midnight, it starts raining. A **flood begins**, and the water rises **uniformly and steadily** around the island.

You are standing at a **random point on the island**. You know:
– The island is **symmetrical and flat**, like a giant disk.
– The water will **reach the center in exactly 60 minutes**.
– You can walk at a **constant speed**, and your **goal is to survive** by reaching the **edge of the island** before the water cuts you off.

· But here's the twist:
You are **blindfolded** and **don't know which direction** leads to the edge.

• **Question:**
What is the optimal strategy to guarantee your survival?
Also, **how many minutes will it take in the worst-case scenario?**
· Assume your walking speed is such that you can **walk straight from center to edge in 60 minutes** if you knew the direction - but remember: you don't.

1. The Solution

Walk in an outward spiral path - specifically, an Archimedean spiral, where the radius increases linearly with the angle you've walked.

2. Intuition

• **Spiral Path Strategy:**
– Start walking in a **logarithmic spiral** outward from the center.
– This spiral ensures that **no matter what direction you're facing**, you will **eventually reach the edge** - without ever needing to know which way you're going.

• **Worst-case Time Taken:**

The worst-case time required with this strategy is approximately:

2.278 minutes per unit of radius $\times$ island's radius
Since you can walk from center to edge in 60 minutes, your radius unit is 1 $\rightarrow$ So:
$2.278 \times 60 \approx 137$ minutes

So the worst-case time it takes to reach the edge using the spiral path is about 137 minutes.

That's over twice as long - but it's the only deterministic survival strategy if you're blindfolded and disoriented.

$\Rightarrow$ **Why Spiral Works:**

Unlike the straight-line method (which can fail catastrophically), the spiral:
· **Always moves outward,**
· **Sweeps every direction,**
· **Eventually reaches the boundary, no matter which way you started.**

Yes - it takes up to **137 minutes**, but because the spiral grows **outward**, you'll **reach the edge before the water reaches you**, as long as your **radial distance increases faster than the water's**.

In other words: the spiral must be tuned so that the **radial component of your speed** ensures you reach radius = 1 (i.e., the edge) **before t = 60 minutes**.

And that's **exactly** how the spiral is constructed - it spreads outward fast enough so that **you escape before drowning**.

• **So What's the Real Deal?**

i) Straight-line (blindfolded): **Not safe**, takes **up to 120 mins**, and you drown after 60.
ii) Spiral path: **Always safe, guarantees escape** before 60 minutes radius is over-taken - even though **total path length** is $\sim$ 137 mins, the radial escape happens **within 60 minutes**.

Day 24 $\xrightarrow{}$

The Problem Statement :-

Imagine you are given a **circle of 100 chairs**, numbered 1 to 100.
You and 99 others are standing around it.

A game begins:

1. Person #1 **kills** the person to their immediate left (#2), then hands the knife to the next living person (#3).
2. That person kills the next living one (i.e., #4), and so on $\cdots$
3. This process continues - **every second person is eliminated**, wrapping around the circle when necessary - until **only one person remains**.

• **Question:**
You are **person #1**.
To **guarantee survival**, which **numbered position** should you stand in **before the game begins**?

1. The Solution

If you're person #73 in this elimination circle of 100 people, and the game plays out as described (kill every 2nd), **you survive**.

2. Intuition

- **The Josephus Problem (Classic Variant)**

This is a famous theoretical puzzle called the **Josephus Problem**, specifically:

Every **second person** is **eliminated in a circle** until only one remains.

The rule for survival position when counting every 2nd person (i.e., `k = 2`) is:

- **Formula:**

If there are `n` people in the circle, the position of the survivor is:

$$2 \times (n - 2^L) + 1$$

Where 2^L is the largest power of 2 less than or equal to n

$\Rightarrow$ Let's calculate for n = 100:

· Largest power of $2 \leq 100$ is **64**
· So, $\text{L} = 6$ (because $2^6 = 64$)
· Plug into formula:

$2 \times (100 - 64) + 1$
$= 2 \times 36 + 1$
$= 72 + 1$
$= 73$

Day 25 $\xrightarrow{}$

The Problem Statement :-

You're given **two sealed envelopes**.
Each contains **a positive amount of money**, and one of them has **exactly twice as much as** the other.

You're told:
"Pick one envelope. You can open it and look inside. Then decide whether to **keep it or switch**."

You pick one envelope, open it, and find **1,000** inside.

You start reasoning:
"There's a 50% chance this is the smaller amount, so the other envelope has 2,000.
There's also a 50% chance this is the larger amount, so the other one has 500.
So expected value of switching = $(0.5 \times 2{,}000) + (0.5 \times 500) = 1{,}250$
That's more than 1,000... So I should always switch!"

But wait - that logic would make you **keep switching forever**.

• **Question:**
Where is the flaw in this reasoning?
Why is the "expected value" argument **wrong**, even though it looks solid?
Think carefully - no calculations required, just **clarity of thought**.

1. My Thought Process

No, I think when considering expected value, we should factor in profit or loss.
The expected value here is: $(0.5 \times 1000) + (0.5 \times 500) = 750$.
Since the probability of losing 500 is equal to the chance of gaining 1000, the expected value favors keeping.
So, we should choose to keep.

2. Actual Logic

- **Core issue:**

You **cannot assign equal 50/50 probabilities** after seeing 1,000 unless you know the prior distribution of money amounts.

- **Why?**

Let's say the envelopes contain **X and 2X**.
If you open 1,000, then the two possibilities are:

– 1,000 = X → other envelope has 2,000
– 1,000 = 2X → other envelope has 500

But **both scenarios are not equally likely.**

To compute actual probabilities, you'd need to know:

– How likely the game was to include a 500/1,000 pair
– Versus a 1,000/2,000 pair
– That depends on how the amounts were chosen before the game (the prior distribution)

- **The illusion:**

The logic **pretends** to use expected value, but it **treats your 1,000 as a fixed number** without accounting for how likely that amount actually is in the setup.

It's like saying:
"No matter what number I see, switching is better" But that leads to an **infinite loop of switching** - which is irrational.

So:
– The expected value calculation isn't wrong in math
– It's **wrong in context**, because it **misuses probabilities** that depend on unknowns.

Day 26 $\xrightarrow{\hspace{1cm}}$

The Problem Statement :-

You're handed a mysterious calendar.
On this calendar, each **date number** (1 to 31) is replaced by a **3-letter code**. Here's a glimpse:

Date	Code
1	AMN
2	BFO
3	CGP
4	DHQ
5	EIR
6	FJS
7	GKT
8	HLU
9	IMV
10	JNW

You start noticing that each code:
– Seems to **increase letter-by-letter**
– But you can't quite crack the pattern yet

• **Your Task:**
Figure out the rule that maps each **date number** to its **3-letter code**.
Then use your rule to find the code for **date 26**.

1. The Solution

Answer for Day 26: **Z M M**

2. Intuition

• **The Hidden Pattern:**

Let's look at how the letters progress using the **ASCII positions** of characters:

Let's decode each code as three positions:

1. First letter
2. Second letter
3. Third letter

Date	Code	Letters (positions)
1	A M N	(65, 77, 78)
2	B F O	(66, 70, 79)
3	C G P	(67, 71, 80)
4	D H Q	(68, 72, 81)
5	E I R	(69, 73, 82)
6	F J S	(70, 74, 83)
7	G K T	(71, 75, 84)
8	H L U	(72, 76, 85)
9	I M V	(73, 77, 86)
10	J N W	(74, 78, 87)

Now we see the actual rule:

• **Rule:**

For day `n`, the code is:

· **1st letter:** `'A' + (n - 1)`

· **2nd letter:** `'M' + (n - 1)`

· **3rd letter:** `'N' + (n - 1)`

So for `n = 1`:

· 'A' + 0 = A
· 'M' + 0 = M
· 'N' + 0 = N → AMN

For `n = 2`:

· 'A' + 1 = B
· 'M' + 1 = N
· 'N' + 1 = O → But wait, it says BFO instead?
That doesn't match.

BUT here's the twist: the **second and third letters** follow a **different base** than the first one.

Look again at positions:

Date	Code	1st	2nd	3rd
1	A M N	65	77	78
2	B F O	66	70	79
3	C G P	67	71	80
4	D H Q	68	72	81

The second letters go: M, F, G, H, I, J, K, L, M
Which is **cyclically repeating** through **F to N**

So it looks like:
– **1st letter:** A to Z → increment by 1
– **2nd letter:** F to N → cycles every 9
– **3rd letter:** O to Z then wraps

We don't have to fully decode each rule because you **identified the increasing logic** - and based on this...

⇒ **For day 26:**

Let's go from **A** → 1st letter = Z
Now second letter (26th in cycle):
Look back: M(1), F(2), G(3), ..., the cycle from F to N is 9 letters
So index 26 → position (26 % 9) = 8 → 8th letter in [F, G, H, I, J, K, L, M, N] = M

Third letter starts from N (ASCII 78) + 25 = 103 → ASCII 103 = **g**

So final answer = **ZMg** - but that has a **lowercase** letter, which seems off Let's instead assume third letter wraps after Z:
· Start at N (78), +25 = 103 → 103 ¿ 'Z' (90), so we wrap around:
  ('N' + 25 - 26) = 'M'
So third letter is M

Day 27 $\xrightarrow{}$

The Problem Statement :-

You are on an island where:
– **Truth-tellers always tell the truth.**
– **Liars always lie.**

You meet **three people: A, B, and C**.

Each of them says **exactly one sentence**:
1. **A says:** "B is a liar."
2. **B says:** "C is a liar."
3. **C says:** "A and B are liars."

• **Your Task:**
Determine who is the liar and who is the truth-teller among A, B, and C. There is **at least one liar and one truth-teller**.

1. The Solution

After evaluating all possible truth-teller and liar combinations, we deduce:

– **A is a liar.**
– **B is a truth-teller.**
– **C is a liar.**

The key strategy is to assume one person is a truth-teller and trace logical implications of their and others' statements. We discard assumptions that lead to contradictions, leaving the consistent scenario as our final answer.

2. Step-by-Step Analysis

- **Assume C is a truth-teller.**

C says: "A and B are liars."
If C is telling the truth, then:

· A is a liar
· B is a liar

Check if this is consistent with what A and B say:

· A says: "B is a liar."
But A is a liar → this statement is **false** → **B is not a liar** → contradiction!
(We assumed B is a liar, but now we get B is not a liar.)

⇒ **Inconsistency! So C cannot be a truth-teller.**

- **Assume C is a liar.**

C says: "A and B are liars."
If C is lying, the statement is false → "A and B are not both liars."
This means: **at least one of A or B is a truth-teller**.

Let's now test both A and B.

case 1: Try A is a truth-teller:

A says: "B is a liar." → A is truthful → B is a liar.

Check B's statement: "C is a liar."
But B is a liar → the statement is **false** → **C is not a liar** → **contradiction** (we assumed C is a liar)!

⇒ **Inconsistency! So A cannot be a truth-teller.**

case 2: Try B is a truth-teller:

B says: "C is a liar." → B is truthful → C is a liar (matches assumption)

Now check A's statement: "B is a liar."
But A is a liar (we haven't assumed it, but we'll verify).
If A is a liar, then their statement "B is a liar" is false → so B is not a liar → consistent (B is a truth-teller).

All fits!

• **Summary of the Logic**

– C cannot be a truth-teller because that leads to contradiction.
– Assuming C is a liar, and B is a truth-teller leads to a consistent assignment:
 · B says C is a liar $\rightarrow$ true.
 · A (a liar) says B is a liar $\rightarrow$ false $\rightarrow$ B is a truth-teller.
 · C (a liar) says both A and B are liars $\rightarrow$ false $\rightarrow$ at least one of them is a truth-teller $\rightarrow$ satisfied.

Day 28 $\longrightarrow$

The Problem Statement :-

The Missing Number Puzzle:

You are given the following pattern:

$2 \rightarrow 4$
$3 \rightarrow 9$
$4 \rightarrow 16$
$5 \rightarrow 25$
$6 \rightarrow 36$
$7 \rightarrow 49$
$8 \rightarrow 64$
$9 \rightarrow 81$
$10 \rightarrow ?$

Now, that's easy, right? But wait...

Then you're given another version of the same puzzle with a twist:

$2 \rightarrow 4$
$3 \rightarrow 8$
$4 \rightarrow 15$
$5 \rightarrow 24$
$6 \rightarrow 35$
$7 \rightarrow 48$
$8 \rightarrow 63$
$9 \rightarrow 80$
10 ?

• **Your Task:**
Find the missing number for 10 in the second pattern.

Don't just look - think! What changed?

1. The Solution

$10 \rightarrow 99$

2. Analysis

• **For first pattern:**

This is clearly:

$$n \to n^2$$
$$i.e. \, 10 \to 100$$

• **For second pattern:**

Let's analyze the differences:

n	Output	Difference from n^2
2	4	$2^2 = 4 \to 0$
3	8	$3^2 = 9 \to$ -1
4	15	$4^2 = 16 \to$ -1
5	24	$5^2 = 25 \to$ -1
6	35	$6^2 = 36 \to$ -1
7	48	$7^2 = 49 \to$ -1
8	63	$8^2 = 64 \to$ -1
9	80	$9^2 = 81 \to$ -1

So from **n = 3 onward**, the pattern is:
$n \to n^2 - 1$

Hence:
$10 \to 10^2 - 1 = 100 - 1 = 99$

⇒ **Summary:**

The trick was in **spotting where the rule changes**:
At $n = 2,$ output is n^2, but from n = 3 onward, it's $n^2 - 1$.

Day 29 $\xrightarrow{\hspace{2em}}$

The Problem Statement :-

A wall clock is broken in such a way that **it runs 6 minutes slow every hour**.

One day, you set the correct time on it at **12:00 noon**.
Later that day, when you look at the clock again, it shows **3:00 PM**.

• **Your Task:**
What is the actual time when the broken clock shows 3:00 PM?

Give your answer in **HH:MM format**.

1. My Thought Process

I initially assumed that if the clock lags by 6 minutes every hour, then when the clock shows 3:00, the actual time could be 3:18.

2. The Solution

3:20 PM

So the actual time is **03:20 PM** when the broken clock shows **03:00 PM**.

3. Actual Intuition

• **Clock loses 6 minutes every hour.**

That means:
– For **every 60 minutes** of real time,
– The clock shows only **54 minutes**.

So the clock is running at 90% speed ($54/60 = 0.9$).

When the broken clock shows 3:00 PM, how much real time has passed?

From **12:00 PM to 3:00 PM**, the clock thinks **3 hours** have passed = **180 minutes (broken clock time)**.

Let real time passed = **x** minutes.
We know the broken clock shows **90% of real time**, so:

$$0.9 \times x = 180 \quad (1)$$
$$\Rightarrow x = 200 minutes \quad (2)$$

So **200 minutes of real time** passed after 12:00 PM:
⇒ 200 minutes = 3 hours 20 minutes

• **Another Approach: Compounding Lag using Geometric Series**

Instead of setting up equations, you can understand the time difference using the idea of **compounding lag** - where the clock's delay keeps adding on itself.

– **Key Points:**

1. The clock loses 6 minutes every hour, which means it's running at **90% speed** (i.e., 10% slow).
2. When the broken clock shows **3:00 PM**, it has supposedly run for **3 clock hours**, i.e., **180 minutes**.
3. Over that time, the clock should have lagged **18 minutes** - but:
 · That **18 minutes also occurred under the slow clock**, so the lag during those minutes wasn't accounted for.
4. So we add **10% of 18 minutes**, then **10% of that**, and so on:

$$\textbf{Total lag} = 18 + 1.8 + 0.18 + \cdots$$

5. This forms a **geometric series**:

$$\textbf{Total lag} = \frac{18}{1 - \frac{1}{10}} = \frac{1.8}{0.9} = 20\text{min}$$

6. So the actual time is:
3:00 PM + 20 minutes = 3:20 PM

– **Why This Works:**
You're accounting for the **compounding nature** of lag - each correction brings in a bit more error, just like interest compounding over time.

Day 30 $\longrightarrow$

The Problem Statement :-

The Setup:
– There are **100 prisoners**, each kept in solitary confinement.
– There is a **single room** with a **light bulb** (initially off).
– The warden will **randomly pick one prisoner** each day to visit the light bulb room. He can choose the **same prisoner multiple times**, and the others won't know who visited when.
– Prisoners can **toggle** the light bulb (on → off or off → on) or leave it unchanged. – They can use the light bulb as a tool to **communicate indirectly**. – At any point, **any prisoner can declare that all 100 prisoners have visited the room at least once**.

If they're **correct**, everyone is set free.
If they're **wrong**, all are **executed**.

• **Your Task:**
Devise a **strategy** the prisoners could agree upon **before** the visits start, to **guarantee** freedom eventually - **no guessing**, just logic.

You don't need to code - just explain the **core idea** of the strategy clearly.

1. My Thought Process

At first, I considered the case where a new person goes for the first time, and the light is either turned on, off, or turned on again so that the next person sees a fading light — but none of these approaches seemed to work.

2. Core Idea

• **Strategy (agreed beforehand):**

1. Choose **one prisoner** as the **"counter"** - the only one who is responsible for tracking how many unique prisoners have visited.
2. The other **99 prisoners** are **"non-counters."**

Light Bulb Communication Rules:

• **For Non-Counters:**

– Each non-counter **remembers whether they've ever turned ON the bulb**.
– If they **enter the room and the light is OFF**, and they **have never turned it ON before**, they **turn it ON** exactly **once** in their lifetime.
– If they already turned it ON once, or the bulb is ON when they enter - **do nothing**.

• **For the Counter:**

– Every time the **counter enters and finds the light ON**, they:
  · Turn it OFF.
  · Increase their mental count by 1.

• **How It Works:**

– Every time a non-counter gets to the room for the **first time**, and the bulb is OFF, they signal their visit by turning it ON.
– Eventually, the counter will visit and see the bulb ON - and know **someone new** has visited.
– After the counter has turned the bulb OFF **99 times**, they know that all 99 other prisoners have visited.

At that point, the counter can **declare confidently** that all prisoners have visited.

• **Edge Cases Covered:**

– Even if some prisoners are picked repeatedly early on, the system still works.
– Patience is key - the process may take **many days**, but it is **guaranteed to finish eventually.**

$\Rightarrow$ **Final Declaration:**

Once the **counter reaches 99**, they declare:
  "All 100 prisoners have visited the room!"
Everyone is freed.

`--- End of Challenge ---`