Prefix firewall

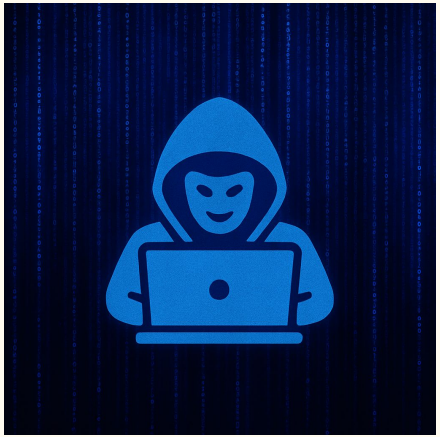
Ad-Hoc Mathematical Puzzle

Cyclic-prefix divisible numbers

Problem Statement

→ You are a grey-hat hacker trying to identify security vulnerabilities in system firewalls. Each system you probe protects itself with a numeric firewall:

1. A list of passcodes, one per line
2. The system is considered secure only if every passcode satisfies a special cyclic prefix-divisibility rule:



Problem Statement

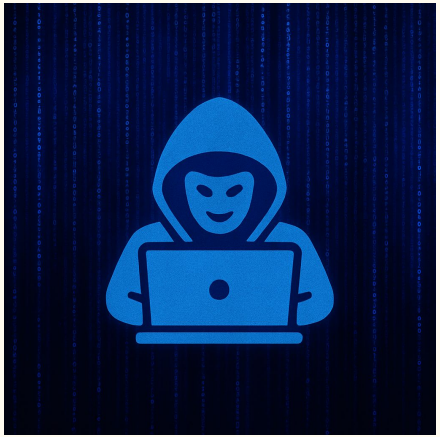
Let $s = s_1 s_2 \ldots s_d$ be the decimal digits of a positive integer. For each prefix length k ($1 \leq k \leq d$), let p_k be the integer formed by the first k digits. The firewall uses a repeating modulus cycle to check divisibility:

$$1, 2, 3, \ldots, 9, 10, 1, 2, \ldots$$

so the modulus m_k for prefix p_k is such that $m_k = ((k - 1) \bmod 10) + 1$

A passcode is considered cyclic-polydivisible if and only if

$$p_k \bmod m_k = 0 \text{ for all } k = 1, 2, \ldots, d$$



Metadata

- Difficulty = **3.7 - Medium**
  - Quadrant = 1.2
  - Prime Path = 2.0
  - Watering Grass = 3.9

Input/Output Specifications

Input

Each input consists of one test case, representing a single secure system.

The first line contains an integer n ($1 \leq n \leq 1\ 000$), the number of passcodes in the firewall.

Each of the next n lines contains one passcode, a positive integer consisting of between 1 and 4 300 digits (inclusive).

Output

If every passcode line satisfies the cyclic prefix-divisibility rule defined above, output that the system is secure.

If the firewall is not secure, print “not secure” followed by each non cyclic prefix-divisible passcode, in the order in which they appear.

Sample inputs

Sample Input 1	Sample Output 1
4 38165472 4 666450405060 26	secure

Sample Input 2	Sample Output 2
4 325 381654729 63 8	not secure 325 63

Sample Input 3	Sample Output 3
3 1 38 12	secure

Secret inputs

- Large n
- (Very) large L
- All non-cyclic-prefix divisible
- Near-polydivisible numbers
- Passing public test for wrong reason

Input Validator

```python
import sys
import re

n_line = sys.stdin.readline()
print(repr(n_line))
assert re.match('^[1-9][0-9]*\n$', n_line)
n = int(n_line)
assert 1 <= n <= 1000

for _ in range(n):
    case_line = sys.stdin.readline()
    print(repr(case_line))
    assert re.match('^[1-9][0-9]{0,4299}\n$', case_line)

assert sys.stdin.readline() == ''

sys.exit(42)
```

- n is an int with no leading zeros, $1 \leq n \leq 1\,000$
- Subsequent passcodes are positive ints, ≤ 4300 digits long
- No additional input

Equivalent in checktestdata validation language:

```
INT(1,1000,n) NEWLINE
REPI(j,n)
    # passcode: 1-25 digits, no leading zeros
    REGEX("[1-9][0-9]{0,4299}", s[j]) NEWLINE
END

EOF
```

Different Solutions

1: Standard Prefix Calculation

- Lines 1 - 8 are equivalent across all solutions
- For all lines until EOF:
  - **Parse every prefix using string slicing with incremental i**
  - **Convert parsed String into Integer**
  - Calculate m as stated in problem description
  - Check if the whole prefix is divisible by m
- Print final results

Results in TLE with $O(n * L^2)$

```python
import sys

data = sys.stdin.read().strip().split()
n = int(data[0])

idx = 1
bad = []

for _ in range(n):
    s = data[idx]
    idx += 1

    ok = True
    for i in range(1, len(s) + 1):
        prefix = s[:i]
        val = int(prefix)
        m = ((i - 1) % 10) + 1
        if val % m != 0:
            ok = False
            break

    if not ok:
        bad.append(s)

if not bad:
    print("secure")
else:
    print("not secure")
    print("\n".join(bad))
```

2: LCM of 0,1,2...10

- ➢ Lines 1 - 8 are equivalent across all solutions
- ➢ For all lines until EOF:
  - **Initiate a "residual" prefix which we will update as so:**
    - **Multiply by 10 and add current int. Mod the total by 2520 (LCM of 0,1,2,...,10). This only leaves us with a reduced remainder $\leqslant 2520$**
    - **Then we check that the remainder is divisible by the current m, which is the case only if it satisfies the cyclic divisibility condition**
    - **Because 2520 is the LCM of all moduli in the cycle, divisibility by any required m is preserved when the prefix is reduced modulo 2520**
- ➢ Print final results

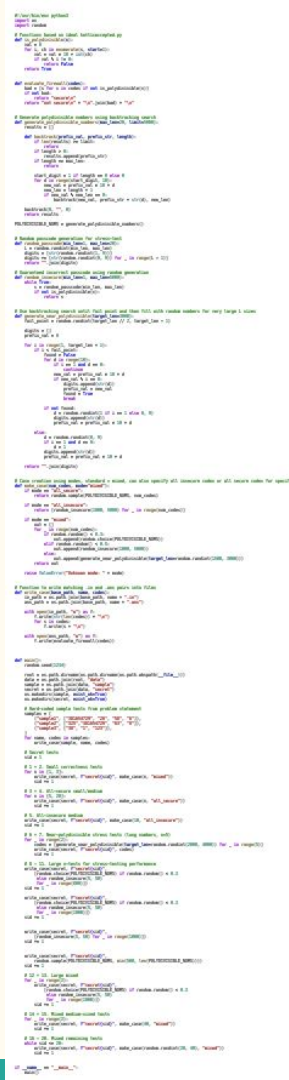
Results in linear parsing of string → O(n * L)

```python
import sys

data = sys.stdin.read().strip().split()
n = int(data[0])

idx = 1
bad = []

for _ in range(n):
    s = data[idx]
    idx += 1

    residual = 0
    ok = True
    for i, ch in enumerate(s, start=1):
        residual = (residual * 10 + int(ch)) % 2520
        m = ((i - 1) % 10) + 1
        if residual % m != 0:
            ok = False
            break

    if not ok:
        bad.append(s)

if not bad:
    print("secure")
else:
    print("not secure")
    print("\n".join(bad))
```

Test Case Generator

Core logic (taken from accepted solution)

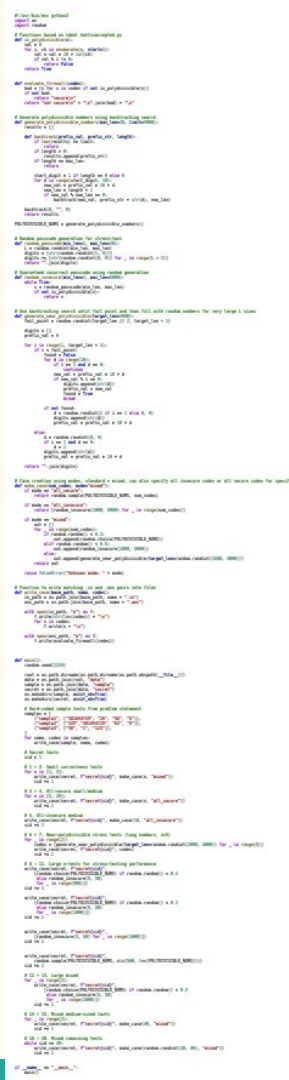
- m_k(k)
  - Computes required modulus for k-th digit
- is_cyclic_polydivisible(s)
  - Function to decide whether a single passcode s is cyclic-polydivisible
  - Returns True if all prefixes pass; False otherwise
- evaluate_firewall(codes)
  - Compute the expected output for a list of passcodes (one test case)
  - Filters out all codes that are not cyclic-polydivisible using is_cyclic_polydivisible
  - If none are bad, return **secure**, otherwise returns **not secure**, followed by each code



Test Case Generator

Generating polydivisible numbers for testing

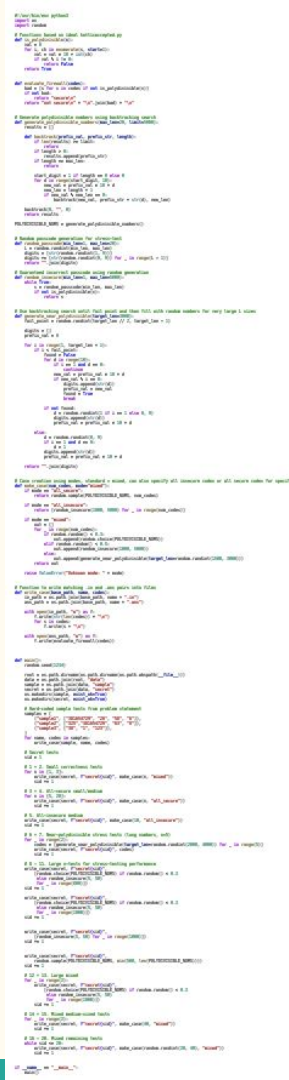
- generate_polydivisible_numbers(max_len=900, limit=1000)
  - Uses backtracking to build all polydivisible numbers up to **`max_len`**, but stops once it has **`limit`** results. → Stores in a large array POLYDIVISIBLE_NUMS
- random_passcode(min_len=1, max_len=4300)
  - Generate a guaranteed non-cyclic polydivisible passcode to be included in the test cases
- generate_near_polydivisible(target_len=4300)
  - Make very long codes that start out polydivisible and then break later, to stress-test prefix logic and efficiency
- random_insecure(min_len=1, max_len=4300)
  - Generate a passcode that is guaranteed not cyclic-polydivisible



Test Case Generator

Creating cases using predefined polydivisible numbers vs random numbers:

- make_case(num_codes, mode="mixed")
  - Different modes (mixed is standard)
    - “All-secure” : Only uses POLYDIVISIBLE_NUMS as argument to num_codes
    - “All-insecure” : Uses random_passcode() call to generate guaranteed non-polydivisible number (additional check implemented)
    - “Mixed” : Uses Random.random() to alternate between secure and non-secure
    - One hand-made test: cyclic-poly divisible number of max length for an int



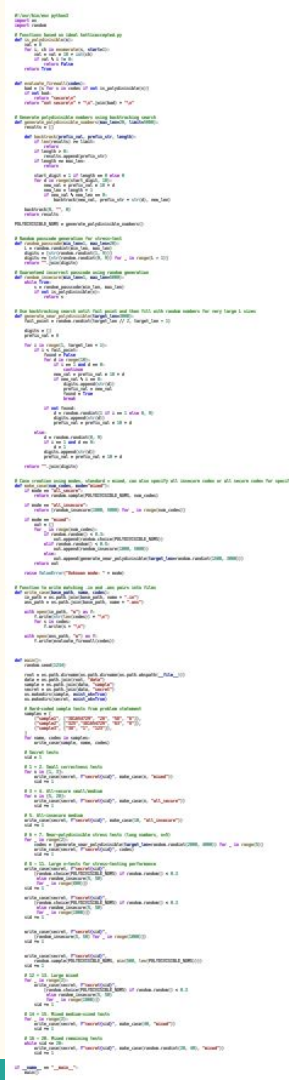
Test Case Generator

Writing to Case files with incrementing id's:

- write_case(base_path, name, codes)
  - Redirects make_case(num_codes, mode) output to .in files whilst also calling evaluate_firewall(codes) and directing its output into a new .ans file, using an increment i to label the uid of the cases and ensure they match

```
sample = os.path.join(data, "sample")
secret = os.path.join(data, "secret")
```

- Using paths, we can call write_case, with make_case() as its argument for codes given a mode, which will reflect the name we specify (in this case, we just named each test secret**i** where i was incremented to 20)



EOF