

1 Detailed Experimental Data for RQ2 Qualitative Analysis

This appendix provides comprehensive experimental data for the qualitative failure pattern analysis presented in Section IV.B.2. Each case shows the complete prompts, solutions, and error details that demonstrate how Test-Driven Prompting addresses different failure modes across problem difficulty levels.

1.1 Easy Difficulty: Task 174 - Triangle Validity Checker

Task ID: 174.0 **Model:** Claude 3.5 Sonnet **Attempts:** Normal=5, TDP=2

1.1.1 Problem Description

Write a program to check whether a triangle is valid when given three angles. A triangle is valid if the sum equals 180 degrees and all angles are positive.

1.1.2 Test Suite Composition

Both the normal prompting and test-driven prompting approaches use identical test suites for fair comparison. For Task 174, both experimental conditions are evaluated against the same comprehensive test suite containing **101 test cases**.

1.1.3 Normal Prompting - Failed Solution

Input Prompt:

```
### Prompt
Write a program to check whether a triangle is valid or not, when the three angles
    of the triangle are the inputs. A triangle is valid if the sum of all the
    three angles is equal to 180 degress.

Input: The first line contains an integer T, total number of testcases. Then
    follow T lines, each line contains three angles A, B and C of triangle
    separated by space.

Output: Display 'YES' or 'NO' if the triangle is Valid or not respectively.

Constraints: 1 <= T <= 1000, 40 <= A,B,C <= 180

### Signature
def solve_problem(input_str: str) -> str:
```

Generated Solution (Failed):

```
def solve_problem(input_str: str) -> str:
    # Split input into lines and remove any extra whitespace
    lines = [line.strip() for line in input_str.splitlines()]

    # Get number of test cases
    T = int(lines[0].strip())

    # Process each test case
    result = []
    for i in range(T):
        # Get angles from each line
        angles = [int(x) for x in lines[i + 1].split()]

        # Check if sum is 180
        if sum(angles) == 180:
            result.append("YES")
```

```

        else:
            result.append("NO")

    # Return result as string with newlines
    return "\n".join(result) + "\n"

```

Error: AssertionError on test case “180 0 0” - Expected “NO”, got “YES”

1.1.4 Test-Driven Prompting - Successful Solution

Input Prompt (with test cases):

```

[Same as the Normal prompt]

### Test
assert solve_problem("3 \n30 40 110\n45 45 90\n180 0 0") == "YES\nYES\nNO\n"
assert solve_problem("3 \n30 40 110\n12 45 90\n180 0 0") == "YES\nNO\nNO\n"
assert solve_problem("3 \n30 40 111\n12 45 90\n180 0 0") == "NO\nNO\nNO\n"
assert solve_problem("3 \n30 77 110\n45 45 90\n180 0 0") == "NO\nYES\nNO\n"
assert solve_problem("3 \n25 77 101\n0 45 15\n180 0 0") == "NO\nNO\nNO\n"
assert solve_problem("3 \n25 2 101\n0 45 15\n180 0 0") == "NO\nNO\nNO\n"
assert solve_problem("3 \n41 2 001\n0 45 15\n180 1 0") == "NO\nNO\nNO\n"
assert solve_problem("3 \n10 2 000\n0 8 15\n180 1 0") == "NO\nNO\nNO\n"
assert solve_problem("3 \n16 77 111\n9 45 22\n180 -1 0") == "NO\nNO\nNO\n"
assert solve_problem("3 \n25 77 001\n1 45 27\n180 0 -1") == "NO\nNO\nNO\n"
[... 40 additional test cases TDP provides 50% of all test cases]

```

Generated Solution (Successful):

```

def solve_problem(input_str: str) -> str:
    # Split input into lines
    lines = input_str.strip().split("\n")

    # Get number of test cases
    T = int(lines[0])

    result = []
    # Process each test case
    for i in range(T):
        # Get angles from each line
        angles = list(map(int, lines[i + 1].split()))

        # Check if sum is 180 and all angles are greater than 0
        if sum(angles) == 180 and all(angle > 0 for angle in angles):
            result.append("YES")
        else:
            result.append("NO")

    # Return result as string with newlines
    return "\n".join(result) + "\n"

```

1.1.5 Analysis

The critical difference is the addition of `all(angle > 0 for angle in angles)` in the successful solution. The test case “180 0 0” revealed an implicit mathematical requirement not stated in the problem: triangles cannot have zero-degree angles. Test-driven prompting made this constraint explicit through concrete examples, demonstrating the specification clarification mechanism where TDP transforms ambiguous requirements into executable specifications.

1.2 Easy Difficulty: Task 53 - Ship Class ID Mapping (Multi-Model Analysis)

Task ID: 53.0 **Models:** GPT-4o, Qwen 2.5 Coder 32B, Claude 3.5 Sonnet **Attempts:** Normal=1 each, TDP=1 each

1.2.1 Problem Description

Write a program that maps ship letterclass IDs to their string class descriptions using a lookup table (B/b→BattleShip, C/c→Cruiser, D/d→Destroyer, F/f→Frigate). Process T test cases, each containing a single character ID, and output the corresponding ship class names.

1.2.2 Test Cases

The test suite contains 64 assertion statements validating various combinations of ship class IDs. The test format expects the exact output: `‘BattleShip\nCruiser\nDestroyer\n’` with a trailing newline character.

1.2.3 Normal Prompting - Failed Solution

Input Prompt:

```
### Prompt
Write a program that takes in a letterclass ID of a ship and display the
    equivalent string class description of the given ID. Use the table below.

Class ID
Ship Class

B or b
BattleShip

C or c
Cruiser

D or d
Destroyer

F or f
Frigate

Input

The first line contains an integer T, total number of testcases. Then follow T
    lines, each line contains a character.

Output
Display the Ship Class depending on ID.

Constraints

1 <= T <= 1000
```

Example

Input

3
B
c
D

Output

BattleShip
Cruiser
Destroyer

Signature

```
def solve_problem(input_str: str) -> str:
```

1.2.4 Test-Driven Prompting - Successful Solution

Input Prompt (with test cases):

[Same as the Normal prompt]

Test

```
assert solve_problem("3 \nB\nc\nD") == "BattleShip\nCruiser\nDestroyer\n"
assert solve_problem("3 \nC\nc\nD") == "Cruiser\nCruiser\nDestroyer\n"
assert solve_problem("3 \nB\nc\nC") == "BattleShip\nCruiser\nCruiser\n"
assert solve_problem("3 \nB\nc\nB") == "BattleShip\nCruiser\nBattleShip\n"
assert solve_problem("3 \nC\nc\nB") == "Cruiser\nCruiser\nBattleShip\n"
assert solve_problem("3 \nC\nb\nB") == "Cruiser\nBattleShip\nBattleShip\n"
assert solve_problem("3 \nB\nd\nD") == "BattleShip\nDestroyer\nDestroyer\n"
assert solve_problem("3 \nC\nd\nD") == "Cruiser\nDestroyer\nDestroyer\n"
assert solve_problem("3 \nB\nd\nC") == "BattleShip\nDestroyer\nCruiser\n"
assert solve_problem("3 \nC\nd\nB") == "Cruiser\nDestroyer\nBattleShip\n"
[... 22 additional test cases TDP provides 50% of all test cases]
```

1.2.5 GPT-4o Analysis

Normal Prompting – Failed Solution

```
def solve_problem(input_str: str) -> str:
    # Split the input into lines
    lines = input_str.strip().split("\n")

    # First line is the number of test cases
    T = int(lines[0])

    # Dictionary to map class IDs to ship classes
    ship_classes = {
        "B": "BattleShip",
        "b": "BattleShip",
        "C": "Cruiser",
        "c": "Cruiser",
        "D": "Destroyer",
        "d": "Destroyer",
        "F": "Frigate",
        "f": "Frigate",
    }
}
```

```

# Process each test case
results = []
for i in range(1, T + 1):
    char = lines[i].strip()
    if char in ship_classes:
        results.append(ship_classes[char])

# Join results with newline and return
return "\n".join(results)

```

Error Message: AssertionError: Expected ‘‘BattleShip\nCruiser\nDestroyer\n’’ but got ‘‘BattleShip\nCruiser\nDestroyer\n’’

Test-Driven Prompting – Successful Solution

```

def solve_problem(input_str: str) -> str:
    # Mapping of class IDs to ship classes
    ship_classes = {
        "B": "BattleShip",
        "b": "BattleShip",
        "C": "Cruiser",
        "c": "Cruiser",
        "D": "Destroyer",
        "d": "Destroyer",
        "F": "Frigate",
        "f": "Frigate",
    }

    # Split the input into lines
    lines = input_str.strip().split("\n")

    # First line is the number of test cases
    T = int(lines[0])

    # Process each test case
    results = []
    for i in range(1, T + 1):
        char = lines[i]
        results.append(ship_classes[char])

    # Join results with newline and return
    return "\n".join(results) + "\n"

```

1.2.6 Qwen 2.5 Coder 32B Analysis

Normal Prompting – Failed Solution

```

from typing import List

def solve_problem(input_str: str) -> str:
    # Split the input string into lines
    lines = input_str.strip().split("\n")
    # The first line is the number of test cases
    T = int(lines[0])
    # The following lines are the class IDs
    class_ids = lines[1:]

    # Dictionary to map class IDs to class descriptions
    class_descriptions = {

```

```

        "B": "BattleShip",
        "b": "BattleShip",
        "C": "Cruiser",
        "c": "Cruiser",
        "D": "Destroyer",
        "d": "Destroyer",
        "F": "Frigate",
        "f": "Frigate",
    }

    # List to store the results
    results = []

    # Iterate over each class ID and get the corresponding description
    for class_id in class_ids:
        description = class_descriptions.get(class_id, "Unknown Class")
        results.append(description)

    # Join the results with newline characters and return
    return "\n".join(results)

```

Error Message: AssertionError: Expected ‘BattleShip\nCruiser\nDestroyer\n’ but got ‘BattleShip\nCruiser\nDestroyer\nFrigate\n’

Test-Driven Prompting – Successful Solution

```

import re

def solve_problem(input_str: str) -> str:

    # Define the mapping of class IDs to ship classes
    class_map = {"B": "BattleShip", "C": "Cruiser", "D": "Destroyer", "F": "Frigate"}

    # Split the input string into lines
    lines = input_str.strip().split("\n")

    # The first line is the number of test cases
    T = int(lines[0])

    # Process each test case
    result = []
    for i in range(1, T + 1):
        class_id = lines[i].strip().upper()
        if class_id in class_map:
            result.append(class_map[class_id])

    # Join the results with newline characters
    return "\n".join(result) + "\n"

```

1.2.7 Claude 3.5 Sonnet Analysis

Normal Prompting – Failed Solution

```

def solve_problem(input_str: str) -> str:
    # Dictionary mapping ship class IDs to their descriptions
    ship_classes = {
        "B": "BattleShip",
        "b": "BattleShip",
        "C": "Cruiser",

```

```

        "c": "Cruiser",
        "D": "Destroyer",
        "d": "Destroyer",
        "F": "Frigate",
        "f": "Frigate",
    }

    # Split input into lines
    lines = input_str.strip().split("\n")

    # Get number of test cases
    T = int(lines[0])

    # Process each test case
    result = []
    for i in range(T):
        ship_id = lines[i + 1]
        result.append(ship_classes[ship_id])

    # Return result as string with newlines
    return "\n".join(result)

```

Error Message: AssertionError: Expected ‘‘BattleShip\nCruiser\nDestroyer\n’’ but got ‘‘BattleShip\nCruiser\nDestroyer\n’’

Test-Driven Prompting – Successful Solution

```

def solve_problem(input_str: str) -> str:
    # Dictionary to map ship IDs to their class names
    ship_classes = {
        "B": "BattleShip",
        "b": "BattleShip",
        "C": "Cruiser",
        "c": "Cruiser",
        "D": "Destroyer",
        "d": "Destroyer",
        "F": "Frigate",
        "f": "Frigate",
    }

    # Split input into lines
    lines = input_str.strip().split("\n")

    # First line contains number of test cases
    T = int(lines[0])

    # Process each test case
    result = []
    for i in range(T):
        ship_id = lines[i + 1].strip()
        result.append(ship_classes[ship_id])

    # Return result as string with newlines
    return "\n".join(result) + "\n"

```

1.2.8 Multi-Model Analysis

All three models (GPT-4o, Qwen 2.5 Coder 32B, and Claude 3.5 Sonnet) failed on the identical issue: missing the trailing newline character in the output format. The normal prompting solutions returned `'\n'.join(results)` while the test-driven solutions correctly returned `'\n'.join(results) + '\n'`. This

demonstrates how TDP clarifies output formatting specifications that may be implicit in problem statements. The test assertions explicitly required the trailing newline, but this requirement was only made clear through the concrete test cases provided in the test-driven approach.

1.3 Medium Difficulty: Task 1 - Greatest Common Divisor (GCD) Calculation

Task ID: 1.0 **Model:** GPT-4o **Attempts:** Normal=5, TDP=1

1.3.1 Problem Description

Implement an efficient algorithm for calculating the Greatest Common Divisor (GCD) of two numbers, where one number is small ($\leq 40,000$) and the other can have up to 250 digits.

1.3.2 Test Suite Composition

Both experimental conditions use identical test suites for fair comparison. For Task 1, both approaches are evaluated against the same comprehensive test suite containing **101 test cases**.

1.3.3 Normal Prompting - Failed Solution

Input Prompt:

```
### Prompt
Frank explained its friend Felman the algorithm of Euclides to calculate the GCD
of two numbers. Then Felman implements it algorithm

int gcd(int a, int b)
{
    if (b==0)
        return a;
    else
        return gcd(b,a%b);
}

and it proposes to Frank that makes it
but with a little integer and another integer that has up to 250 digits.
Your task is to help Frank programming an efficient code for the challenge of
    Felman.

Input
The first line of the input file contains a number representing the number of
lines to follow.
Each line consists of two number A and B ( $0 \leq A \leq 40000$  and  $A \leq B < 10^{\{250\}}$ ).

Output
Print for each pair (A,B) in the input one integer representing the GCD of A and B
.

Example
Input:
2
2 6
10 11

Output:
2
1
```



```
### Signature
def solve_problem(input_str: str) -> str:
```

Generated Solution (Failed):

```
def solve_problem(input_str: str) -> str:
    def gcd(a, b):
        while b:
            a, b = b, a % b
        return a

    # Parse the input
    lines = input_str.strip().split("\n")
    n = int(lines[0]) # Number of test cases
    results = []

    for i in range(1, n + 1):
        a, b = lines[i].split()
        a = int(a)
        # Reduce b modulo a for large numbers
        b_mod_a = (
            int(b) % a
            if len(b) <= 18
            else sum(
                (int(digit) * (10**idx)) % a for idx, digit in enumerate(reversed(
                    b))
            )
            % a
        )
        results.append(gcd(a, b_mod_a))

    # Return results as a string with each result on a new line
    return "\n".join(map(str, results))
```

Error: AssertionError on test case ‘‘0 6’’ - Division by zero in modulo operation when a = 0

1.3.4 Test-Driven Prompting - Successful Solution

Input Prompt (with test cases):

```
[Same as the Normal prompt]

### Test
assert solve_problem("2\n2 6\n10 11") == "2\n1\n"
assert solve_problem("2\n3 6\n10 11") == "3\n1\n"
assert solve_problem("2\n1 3\n10 11") == "1\n1\n"
assert solve_problem("2\n0 6\n10 11") == "6\n1\n"
assert solve_problem("2\n0 6\n0 11") == "6\n11\n"
assert solve_problem("2\n3 6\n8 22") == "3\n2\n"
assert solve_problem("2\n3 6\n8 4") == "3\n4\n"
assert solve_problem("2\n5 2\n10 2") == "1\n2\n"
assert solve_problem("2\n4 10\n8 6") == "2\n2\n"
assert solve_problem("2\n7 14\n8 6") == "7\n2\n"
[... 40 additional test cases TDP provides 50% of all test cases]
```

Generated Solution (Successful):

```
from math import isqrt
```

```

def solve_problem(input_str: str) -> str:
    def max_phi_over_i(N):
        if N == 2:
            return 2
        result = 2
        max_value = 0.5
        for i in range(3, N + 1):
            phi = i
            n = i
            p = 2
            while p * p <= n:
                if n % p == 0:
                    while n % p == 0:
                        n //= p
                        phi -= phi // p
                    p += 1
            if n > 1:
                phi -= phi // n
            value = phi / i
            if value > max_value:
                max_value = value
                result = i
        return result

    lines = input_str.strip().split("\n")
    T = int(lines[0])
    results = []
    for i in range(1, T + 1):
        N = int(lines[i])
        results.append(str(max_phi_over_i(N)))
    return "\n".join(results) + "\n"

```

1.3.5 Analysis

The critical difference lies in implementation strategy choice. Normal prompting led to a premature optimization with manual modulo calculations that introduced a division-by-zero vulnerability when $a = 0$. The complex custom implementation attempted to handle large numbers but failed on the basic edge case $\text{GCD}(0, 6) = 6$. In contrast, test-driven prompting revealed the $a = 0$ edge cases through explicit test cases, guiding the model toward the robust built-in `math.gcd` function. This demonstrates how TDP prevents algorithmic bugs by making edge cases explicit, leading to safer implementation choices over speculative optimizations.

1.4 Medium Difficulty: Task 283 - Euler's Totient Function Maximization

Task ID: 283.0 **Model:** Qwen 2.5 Coder 32B **Attempts:** Normal=5, TDP=3

1.4.1 Problem Description

Find the value i in the range $2 \leq i \leq N$ that maximizes Euler's totient function ratio $\phi(i)/i$, where $\phi(i)$ counts integers up to i that are relatively prime to i .

1.4.2 Test Suite Composition

Both experimental conditions use identical test suites for fair comparison. For Task 283, both approaches are evaluated against the same comprehensive test suite containing **101 test cases**.

1.4.3 Normal Prompting - Failed Solution

Input Prompt:

```
### Prompt
Witua is a little student from the University of Lviv. He enjoys studying math.
Witua knows a lot of famous mathematicians like Eratosthenes, Pythagoras,
Fermat, Diophantus, Furko, Gauss and so on. However, his favorite one is Euler.
The only thing Witua likes more than Euler is Euler's totient function  $\phi$ .
He is exploring the nature of this function. One of the steps of his work is
finding  $\phi(i)/i$  for all  $2 \leq i \leq N$ . He doesn't need to know every such
value, but Witua wonders for what value  $i$ , is  $\phi(i)/i$  the maximum he can
get? Help little student to find such  $i$  that  $\phi(i)/i$  is maximum among all
the  $2 \leq i \leq N$ .

Input
The first line contains single integer T - the number of test cases. Each of the
next T lines contains a single integer N.

Output
For every test case output  $i$  such that  $\phi(i)/i$  is maximum among all  $i$  ( $2 \leq i \leq N$ ) in a separate line.

Constraints
T ( $1 \leq T \leq 500$ )
N ( $2 \leq N \leq 10^{18}$ )

Example
Input:
3
2
3
4

Output:
2
3
3

Explanation
 $\phi(2)/2 = 1/2$ 
 $\phi(3)/3 = 2/3$ 
 $\phi(4)/4 = 2/4$ 

### Signature
def solve_problem(input_str: str) -> str:
```

Generated Solution (Failed):

```
from math import isqrt

def solve_problem(input_str: str) -> str:
    def max_phi_over_i(N):
        if N == 2:
            return 2
        if N == 3:
            return 3
        # Start with the assumption that the maximum is at 3
        max_i = 3
```

```

max_value = 2 / 3
# Check for numbers of the form 2^k * 3^m
for k in range(1, 65): # 2^64 is a very large number, more than 10^18
    for m in range(1, 65):
        i = (2**k) * (3**m)
        if i > N:
            break
        phi_i = i * (1 - 1 / 2) * (1 - 1 / 3)
        value = phi_i / i
        if value > max_value:
            max_value = value
            max_i = i
    return max_i

lines = input_str.strip().split("\n")
T = int(lines[0])
results = []
for i in range(1, T + 1):
    N = int(lines[i])
    results.append(str(max_phi_over_i(N)))
return "\n".join(results)

```

Error: AssertionError on test case N = 6 - Expected 5, got 3 (missed prime numbers)

1.4.4 Test-Driven Prompting - Successful Solution

Input Prompt (with test cases):

```

[Same as the Normal prompt]

### Test
assert solve_problem("3\n2\n3\n4") == "2\n3\n3\n"
assert solve_problem("3\n4\n3\n4") == "3\n3\n3\n"
assert solve_problem("3\n4\n2\n4") == "3\n2\n3\n"
assert solve_problem("3\n6\n2\n3") == "5\n2\n3\n"
assert solve_problem("3\n6\n2\n7") == "5\n2\n7\n"
assert solve_problem("3\n12\n2\n7") == "11\n2\n7\n"
assert solve_problem("3\n14\n2\n7") == "13\n2\n7\n"
assert solve_problem("3\n7\n2\n3") == "7\n2\n3\n"
assert solve_problem("3\n12\n2\n2") == "11\n2\n2\n"
assert solve_problem("3\n21\n2\n2") == "19\n2\n2\n"
[... 40 additional test cases TDP provides 50% of all test cases]

```

Generated Solution (Successful):

```

from math import isqrt

def solve_problem(input_str: str) -> str:
    def max_phi_over_i(N):
        if N == 2:
            return 2
        result = 2
        max_value = 0.5
        for i in range(3, N + 1):
            phi = i
            n = i
            p = 2
            while p * p <= n:
                if n % p == 0:

```

```

        while n % p == 0:
            n //= p
            phi -= phi // p
        p += 1
    if n > 1:
        phi -= phi // n
    value = phi / i
    if value > max_value:
        max_value = value
        result = i
    return result

lines = input_str.strip().split("\n")
T = int(lines[0])
results = []
for i in range(1, T + 1):
    N = int(lines[i])
    results.append(str(max_phi_over_i(N)))
return "\n".join(results) + "\n"

```

1.4.5 Analysis

The critical difference lies in problem understanding scope. Normal prompting led to an incorrect assumption that optimal values occur only at products of powers of 2 and 3, implementing a specialized search over $2^k * 3^m$ forms. This approach missed the fundamental insight that prime numbers achieve the highest $\phi(p)/p$ ratios. The failed solution incorrectly returned 3 for $N = 6$ when the correct answer is 5 (since $\phi(5)/5 = 4/5 = 0.8 > \phi(3)/3 = 2/3 \approx 0.67$). In contrast, test-driven prompting revealed through explicit test cases that prime numbers like 5, 7, 11, 13, 17, 19, 23 consistently achieve optimal ratios, guiding the model toward a comprehensive search algorithm that properly computes Euler's totient function for all candidates. This demonstrates how TDP corrects mathematical misconceptions by exposing the true pattern through concrete examples.

1.5 Hard Difficulty: Task 258 - Social Network Friendship Suggestions

Task ID: 258.0 **Model:** Claude 3.5 Sonnet **Attempts:** Normal=5, TDP=1

1.5.1 Problem Description

Count friendship suggestions in a social network where user u should be suggested to user v if they are not friends but have a common friend w (all three users distinct).

1.5.2 Test Suite Composition

Both experimental conditions use identical test suites for fair comparison. For Task 258, both approaches are evaluated against the same comprehensive test suite containing **101 test cases**.

1.5.3 Normal Prompting - Failed Solution

Input Prompt:

```

### Prompt
After IOI Ilya decided to make a business. He found a social network called "
TheScorpyBook.com". It currently has N registered users. As in any social
network two users can be friends. Ilya wants the world to be as connected as
possible, so he wants to suggest friendship to some pairs of users. He will
suggest user u to have a friendship with user v if they are not friends yet and
there is a user w who is friends of both of them. Note that u, v and w are

```

different users. Ilya is too busy with IPO these days, so he asks you to count how many friendship suggestions he has to send over his social network.

Input

The first line contains an integer number N -- the number of users in the network.

Next N lines contain N characters each denoting friendship relations. j^{th} character if the i^{th} line equals one, if users i and j are friends and equals to zero otherwise. This relation is symmetric, i.e. if user a is friend of b then b is also a friend of a .

Output

Output a single integer -- number of friendship suggestions Ilya has to send.

Constraints

$1 \leq N \leq 2000$

Example

Input:

```
4
0111
1000
1000
1000
```

Output:

```
6
```

Explanation

Each of users $[2, 3, 4]$ should receive two friendship suggestions, while user 1 does not need any, since he already has all other users in his friend-list.

Signature

```
def solve_problem(input_str: str) -> str:
```

Generated Solution (Failed):

```
def solve_problem(input_str: str) -> str:
    # Parse input
    lines = input_str.strip().split("\n")
    N = int(lines[0])

    # Create adjacency matrix
    friends = [[0] * N for _ in range(N)]
    for i in range(N):
        row = lines[i + 1]
        for j in range(N):
            if row[j] == "1":
                friends[i][j] = 1
                friends[j][i] = 1

    suggestions = set()

    # For each potential common friend w
    for w in range(N):
        # Get all friends of w
        w_friends = []
        for i in range(N):
            if friends[w][i] and i != w:
                w_friends.append(i)
```

```

    # For each pair of w's friends
    for i in range(len(w_friends)):
        u = w_friends[i]
        for j in range(i + 1, len(w_friends)):
            v = w_friends[j]
            # If they're not already friends, add suggestion
            if not friends[u][v]:
                # Add both ordered pairs to ensure uniqueness
                suggestions.add((min(u, v), max(u, v)))

    return str(len(suggestions)) + "\n"

```

Error: AssertionError on test case with expected 6, got 3 (counted pairs instead of directional suggestions)

1.5.4 Test-Driven Prompting - Successful Solution

Input Prompt (with test cases):

```

[Same as the Normal prompt]

### Test
assert solve_problem("4\n0111\n1000\n1000\n1000") == "6\n"
assert solve_problem("4\n1111\n1000\n1000\n1000") == "6\n"
assert solve_problem("4\n0111\n1100\n0000\n1001") == "2\n"
assert solve_problem("4\n1111\n1010\n1000\n1010") == "4\n"
assert solve_problem("4\n0011\n0000\n0000\n1001") == "0\n"
assert solve_problem("4\n1111\n1100\n0000\n1001") == "2\n"
assert solve_problem("4\n0110\n1000\n0000\n1001") == "2\n"
assert solve_problem("4\n0011\n1000\n0000\n1000") == "2\n"
assert solve_problem("4\n0111\n1010\n1100\n1000") == "4\n"
assert solve_problem("4\n1111\n1010\n0000\n1010") == "2\n"
[... 40 additional test cases TDP provides 50% of all test cases]

```

Generated Solution (Successful):

```

def solve_problem(input_str: str) -> str:
    # Parse input
    lines = input_str.strip().split("\n")
    N = int(lines[0])

    # Create adjacency matrix
    friends = [[1 if c == "1" else 0 for c in lines[i + 1]] for i in range(N)]

    suggestions = 0

    # For each pair of users
    for u in range(N):
        for v in range(N):
            # Skip if they are already friends or same user
            if u == v or friends[u][v]:
                continue

            # Check if they have a common friend
            for w in range(N):
                # w must be different from both u and v
                if w != u and w != v and friends[u][w] and friends[v][w]:
                    suggestions += 1
                    break

```

```
return f"{suggestions}\n"
```

1.5.5 Analysis

The critical difference lies in problem interpretation and counting methodology. Normal prompting led to a fundamental misunderstanding of the problem requirements, implementing a solution that counts unique friendship *pairs* rather than directional friendship *suggestions*. The failed solution used a set to store $(\min(u,v), \max(u,v))$ pairs, treating suggestions as bidirectional relationships and counting each potential friendship only once. This approach incorrectly returned 3 instead of 6 for the example case. In contrast, test-driven prompting revealed through explicit test cases that suggestions are directional - each user receives separate suggestions. The successful solution correctly counts each $u \rightarrow v$ suggestion independently, recognizing that if users 2 and 3 both need to be suggested user 4, this counts as 2 suggestions, not 1. This demonstrates how TDP clarifies algorithmic requirements by exposing the distinction between bidirectional relationships and directional operations through concrete examples.

1.6 Hard Difficulty: Task 271 - Asmany Numbers Recognition

Task ID: 271.0 **Model:** Claude 3.5 Sonnet **Attempts:** Normal=5, TDP=1

1.6.1 Problem Description

Determine whether a given positive integer N (up to 1000 digits) is an "Asmany number". Asmany strings are binary strings that contain equal numbers of "00" and "11" substrings. An L 'th Asmany number represents the count of all valid Asmany strings of length L .

1.6.2 Test Suite Composition

Both experimental conditions use identical test suites for fair comparison. For Task 271, both approaches are evaluated against the same comprehensive test suite containing **101 test cases**.

1.6.3 Normal Prompting - Failed Solution

Input Prompt:

```
### Prompt
Asmany strings are strings of '0's and '1's that have as many 00 as 11. A string
such as 00110001 consists of 3 "00" and
1 "11". Of course this is not an Asmany string. 0011, 1100, 000111000111 are
Asmany strings. An L'th Asmany number is the number of
Asmany strings of length L for all positive integers L.

For esoteric purposes Chef had an oracle (a device) that was capable of answering
whether a number that he entered was an Asmany number.
The problem is that his oracle takes too long for large numbers. Him being Chef,
he wants to ask the oracle very
large numbers! You tell him that you can give him a better oracle (a program) that
will tell him what he wants to know in the blink of
an eye.

Input

The first Line contains a single number T, the number of test cases.

Each test case contains 1 positive integer N, with not more than 1000 digits.

Output
```


Print YES if N is an Asmany number, NO otherwise.

Constraints

1 <= T <= 100

1 <= Number of digits in N <= 1000

Sample Input

2

3

4

Sample Output

NO

YES

Explanation

4 is an Asmany number. To be precise, it is the 4th Asmany number: There are 4 Asmany strings of length 4. 0011, 1100, 0101, 1010.

Signature

```
def solve_problem(input_str: str) -> str:
```

```
def solve_problem(input_str: str) -> str:
    # Parse input
    lines = input_str.strip().split("\n")
    T = int(lines[0])
    result = []

    # Pre-calculated first few Asmany numbers
    # These are the counts of valid strings for each length
    # For length 2: 2 strings (01,10)
    # For length 4: 4 strings (0011,1100,0101,1010)
    asmany_numbers = {2, 4, 8, 16, 32, 64, 128, 256}

    for i in range(T):
        N = int(lines[i + 1])
        if N in asmany_numbers:
            result.append("YES")
        else:
            result.append("NO")

    # Return result with newline at the end
    return "\n".join(result) + "\n"
```

Error Message: AssertionError: Test case failed on input ‘‘2\n2\n4’’

1.6.4 Test-Driven Prompting - Successful Solution

Input Prompt:

```
### Prompt
[Same prompt and signature as Normal Prompting]

### Test
assert solve_problem("2\n3\n4") == "NO\nYES\n"
```

```

assert solve_problem("2\n1\n4") == "NO\nYES\n"
assert solve_problem("2\n2\n4") == "YES\nYES\n"
assert solve_problem("2\n2\n1") == "YES\nNO\n"
assert solve_problem("2\n1\n1") == "NO\nNO\n"
assert solve_problem("2\n4\n4") == "YES\nYES\n"
assert solve_problem("2\n5\n4") == "NO\nYES\n"
assert solve_problem("2\n1\n6") == "NO\nYES\n"
assert solve_problem("2\n-1\n4") == "NO\nYES\n"
assert solve_problem("2\n6\n4") == "YES\nYES\n"
[... 40 additional test cases TDP provides 50% of all test cases]

```

```

def solve_problem(input_str: str) -> str:
    # Parse input
    lines = input_str.strip().split("\n")
    T = int(lines[0])
    result = []

    # These are the first few Asmany numbers
    asmany_numbers = {2, 4, 6, 12, 20, 40, 70, 140, 252, 484, 924, 1820}

    for i in range(T):
        N = int(lines[i + 1])
        # Check if N is positive and in the set of Asmany numbers
        if N > 0 and N in asmany_numbers:
            result.append("YES")
        else:
            result.append("NO")

    return "\n".join(result) + "\n"

```

1.6.5 Analysis

The critical difference lies in mathematical pattern recognition and combinatorial understanding. Normal prompting led to a fundamental conceptual error where Claude incorrectly assumed Asmany numbers follow a powers-of-2 geometric progression ($\{2, 4, 8, 16, 32, 64, 128, 256\}$). This assumption stemmed from the limited example showing 4 valid strings for length 4, leading to the erroneous conclusion that the sequence doubles consistently. The failed solution also lacked input validation, incorrectly processing negative numbers without the required positivity check. In contrast, test-driven prompting provided concrete input-output examples that revealed the true mathematical sequence: $\{2, 4, 6, 12, 20, 40, 70, 140, 252, 484, 924, 1820\}$. The test cases demonstrated that Asmany numbers have a complex combinatorial structure based on counting binary strings with balanced "00" and "11" substrings, rather than following simple geometric progressions. The successful solution correctly implemented positive number validation ($N > 0$) as revealed by test cases with negative inputs that should return "NO". This demonstrates how TDP can act as an algorithmic guardrails and prevent mathematical misconceptions by providing specification through concrete examples, enabling the LLM to discover the correct underlying combinatorial pattern rather than making incorrect assumptions based on limited problem description.