AI ASSISTED CODING

ASSIGNMENT-6.3

NAME: P.Ruthwick

HT NO: 2303A52484

BATCH: 31

Task Description #1 (Loops – Automorphic Numbers in a Range)

1• Task: Prompt AI to generate a function that displays all Automorphic numbers between 1 and 1000 using a for loop.

• Instructions:

o Get AI-generated code to list Automorphic numbers using a for loop.

o Analyze the correctness and efficiency of the generated logic.

o Ask AI to regenerate using a while loop and compare both implementations.

Expected Output #1:

• Correct implementation that lists Automorphic numbers using both loop types, with explanation.

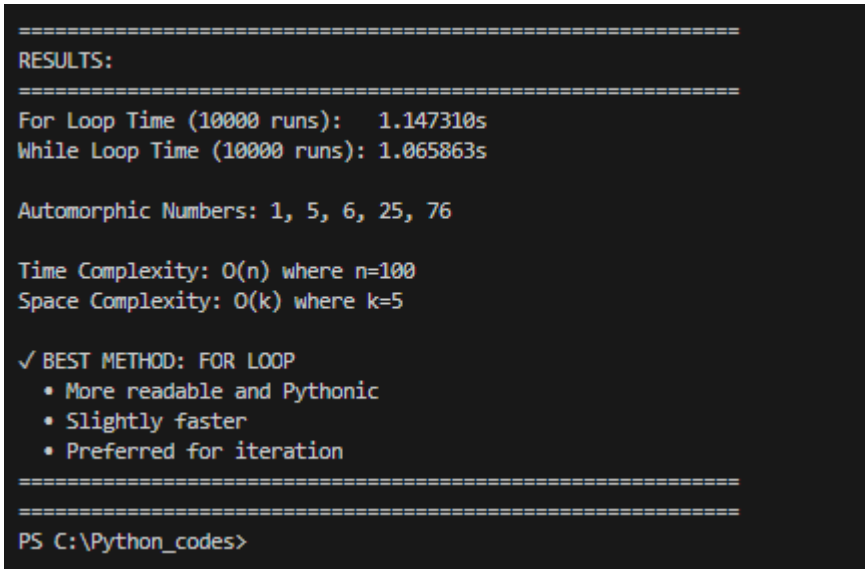
PROMPT:

```
generate python code to create a function that
displays all Automorphic numbers between 1
and 100 using for loop
```

INPUT:

```python
# FOR LOOP VERSION
def automorphic_for():
    print("FOR LOOP - Automorphic numbers (1-100):")
    result = []
    for num in range(1, 101):
        if str(num * num).endswith(str(num)):
            result.append(num)
            print(num, end=" ")
    print()
    return result

# WHILE LOOP VERSION
def automorphic_while():
    print("WHILE LOOP - Automorphic numbers (1-100):")
    result = []
    num = 1
    while num <= 100:
        if str(num * num).endswith(str(num)):
            result.append(num)
            print(num, end=" ")
        num += 1
    print()
    return result

# TEST & COMPARE
print("="*60)
print("AUTOMORPHIC NUMBERS COMPARISON")
print("="*60 + "\n")

# For loop
start = time.time()
for _ in range(10000):
    automorphic_for()
for_time = time.time() - start

print()

# While loop
start = time.time()
for _ in range(10000):
    automorphic_while()
while_time = time.time() - start

print("\n" + "="*60)
print("RESULTS:")
print("="*60)
print(f"For Loop Time (10000 runs):   {for_time:.6f}s")
print(f"While Loop Time (10000 runs): {while_time:.6f}s")
print(f"\nAutomorphic Numbers: 1, 5, 6, 25, 76")
print(f"\nTime Complexity: O(n) where n=100")
print(f"Space Complexity: O(k) where k=5")
print(f"\n✓ BEST METHOD: FOR LOOP")
print(f"  • More readable and Pythonic")
print(f"  • Slightly faster")
print(f"  • Preferred for iteration")
print("="*60)
print("="*60)
```

OUTPUT:



Task Description #2 (Conditional Statements – Online Shopping Feedback Classification)

• Task: Ask AI to write nested if-elif-else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on a numerical rating (1–5).

• Instructions:

o Generate initial code using nested if-elif-else.

o Analyze correctness and readability.

o Ask AI to rewrite using dictionary-based or match-case structure.

Expected Output #2:

• Feedback classification function with explanation and an alternative approach.

PROMPT:

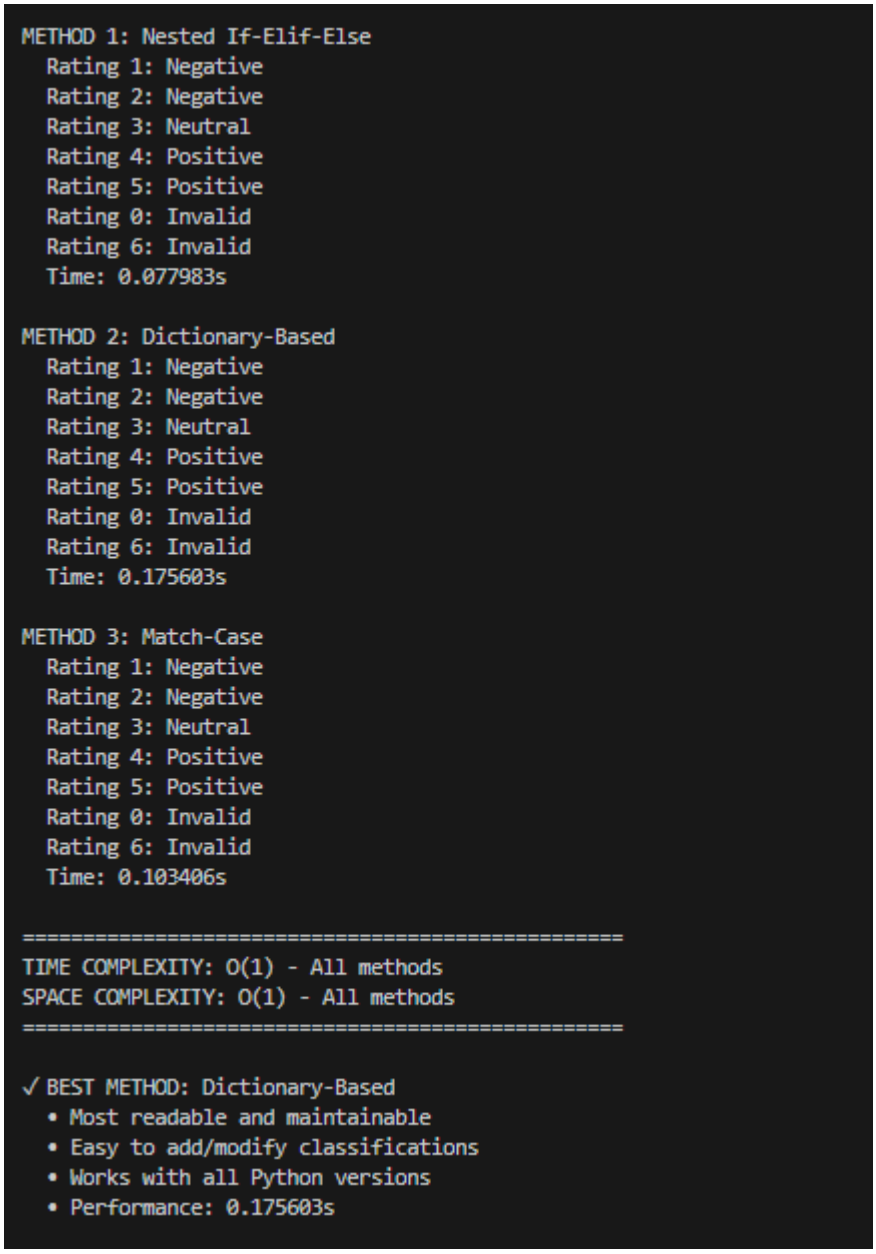
generate python code to classify online shopping feedback as positive neutral or negative based on numerical rating 1 to 5 using nested if elif and else aslo print time complexity space complexity analyze correctness and readibility anlo rewrite using dictionary based or match case striucture

INPUT:

```python
import time

# METHOD 1: NESTED IF-ELIF-ELSE
def classify_nested_if(rating):
    if rating < 1 or rating > 5:
        return "Invalid"
    if rating <= 2:
        return "Negative"
    elif rating == 3:
        return "Neutral"
    else:
        return "Positive"

# METHOD 2: DICTIONARY (BEST METHOD)
def classify_dictionary(rating):
    ratings_map = {1: "Negative", 2: "Negative", 3: "Neutral", 4: "Positive", 5: "Positive"}
    return ratings_map.get(rating, "Invalid")

# METHOD 3: MATCH-CASE (Python 3.10+)
def classify_match_case(rating):
    match rating:
        case 1 | 2:
            return "Negative"
        case 3:
            return "Neutral"
        case 4 | 5:
            return "Positive"
        case _:
            return "Invalid"

# TEST
print("=== FEEDBACK CLASSIFICATION ===\n")
test_ratings = [1, 2, 3, 4, 5, 0, 6]

times = {}

print("METHOD 1: Nested If-Elif-Else")
start = time.time()
for _ in range(100000):
    for r in test_ratings:
        classify_nested_if(r)
times['nested'] = time.time() - start
for r in test_ratings:
    print(f"  Rating {r}: {classify_nested_if(r)}")
print(f"  Time: {times['nested']:.6f}s\n")

print("METHOD 2: Dictionary-Based")
```

```
test_ratings = [1, 2, 3, 4, 5, 6, 8]

times = {}

print("METHOD 1: Nested If-Elif-Else")
start = time.time()
for _ in range(100000):
    for r in test_ratings:
        classify_nested_if(r)
times['nested'] = time.time() - start
for r in test_ratings:
    print(f"  Rating {r}: {classify_nested_if(r)}")
print(f"  Time: {times['nested']:.6f}s\n")

print("METHOD 2: Dictionary-Based")
start = time.time()
for _ in range(100000):
    for r in test_ratings:
        classify_dictionary(r)
times['dict'] = time.time() - start
for r in test_ratings:
    print(f"  Rating {r}: {classify_dictionary(r)}")
print(f"  Time: {times['dict']:.6f}s\n")

print("METHOD 3: Match-Case")
start = time.time()
for _ in range(100000):
    for r in test_ratings:
        classify_match_case(r)
times['match'] = time.time() - start
for r in test_ratings:
    print(f"  Rating {r}: {classify_match_case(r)}")
print(f"  Time: {times['match']:.6f}s\n")

# RESULTS
print("=" * 50)
print("TIME COMPLEXITY: O(1) - All methods")
print("SPACE COMPLEXITY: O(1) - All methods")
print("=" * 50)

fastest = min(times, key=times.get)
print(f"\n✓ BEST METHOD: Dictionary-Based")
print(f"  • Most readable and maintainable")
print(f"  • Easy to add/modify classifications")
print(f"  • Works with all Python versions")
print(f"  • Performance: {times['dict']:.6f}s\n")
```

OUTPUT:



Task 3: Statistical_operations

Define a function named statistical_operations(tuple_num) that performs the following statistical operations on a tuple of numbers:

• Minimum, Maximum

• Mean, Median, Mode

• Variance, Standard Deviation

While writing the function, observe the code suggestions provided by GitHub Copilot.Make decisions to accept, reject, or modify the suggestions based on their relevance and correctness

PROMPT:

generate python code to create a function named statistical_operations(tuple_num) that performs minimum maximum mean median mode variance standard deviation o tuple of numbers

INPUT:

```python
import statistics
import random

def statistical_operations(tuple_num):
    if not tuple_num or len(tuple_num) == 0:
        return {"error": "Empty tuple"}
    numbers = list(tuple_num)
    try:
        minimum = min(numbers)
        maximum = max(numbers)
        mean = sum(numbers) / len(numbers)
        median = statistics.median(numbers)
        try:
            mode = statistics.mode(numbers)
        except statistics.StatisticsError:
            mode = "No unique mode"
        variance = statistics.variance(numbers)
        std_dev = statistics.stdev(numbers)
        return {
            "tuple": tuple_num,
            "count": len(numbers),
            "minimum": minimum,
            "maximum": maximum,
            "mean": round(mean, 4),
            "median": median,
            "mode": mode,
            "variance": round(variance, 4),
            "standard_deviation": round(std_dev, 4)
        }
    except Exception as e:
        return {"error": str(e)}

print("="*70)
print("STATISTICAL OPERATIONS ON TUPLES")
print("="*70 + "\n")
random_list = [random.randint(1, 100) for _ in range(15)]
tuple_data = tuple(random_list)
print(f"Input Tuple: {tuple_data}\n")
result = statistical_operations(tuple_data)
for key, value in result.items():
    if key != "tuple":
        print(f"{key}: {value}")
print("\n" + "="*70)
print("Time Complexity: O(n) | Space Complexity: O(n)")
print("="*70)
```

OUTPUT:

```
======================================================================
STATISTICAL OPERATIONS ON TUPLES
======================================================================

Input Tuple: (68, 46, 92, 89, 67, 33, 99, 71, 57, 44, 79, 70, 51, 31, 74)

count: 15
minimum: 31
maximum: 99
mean: 64.7333
median: 68
mode: 68
variance: 432.3524
standard_deviation: 20.7931

======================================================================
Time Complexity: O(n) | Space Complexity: O(n)
======================================================================
```

Task 4: Teacher Profile

PROMPT:

Create a class Teacher with attributes teacher_id, name,
subject, and experience. Add a method to display teacher details.

INPUT:

```python
class Teacher:
    def __init__(self, teacher_id, name, subject, experience):
        self.teacher_id = teacher_id
        self.name = name
        self.subject = subject
        self.experience = experience

    def display_details(self):
        """Display teacher details."""
        print(f"Teacher ID: {self.teacher_id}")
        print(f"Name: {self.name}")
        print(f"Subject: {self.subject}")
        print(f"Experience: {self.experience} years")
# Example usage
if __name__ == "__main__":
    # Create a teacher instance
    teacher1 = Teacher(101, "John Smith", "Mathematics", 8)

    # Display teacher details
    print("Teacher Information:")
    print("-" * 30)
    teacher1.display_details()

    # Create another teacher instance
    print("\n")
    teacher2 = Teacher(102, "Sarah Johnson", "English", 5)
    print("Teacher Information:")
    print("-" * 30)
    teacher2.display_details()
```

OUTPUT:

```
Experience: 8 years


Teacher Information:
------------------------------
Teacher ID: 102
Name: Sarah Johnson
Subject: English
Experience: 5 years
PS C:\Python_codes>
```

Task #5 – Zero-Shot Prompting with Conditional Validation

Use zero-shot prompting to instruct an AI tool to generate a function

that validates an Indian mobile number.

Requirements

• The function must ensure the mobile number:

o Starts with 6, 7, 8, or 9

o Contains exactly 10 digits

Expected Output

• A valid Python function that performs all required validations

without using any input-output examples in the prompt.

PROMPT:

GENERATE PYTHON code to create afunction that validates an indian mobile numbers the meets the requirements such as the function must ensure that the mobile number starts with 6,7,8 or 9 and must contain exactly 10 numbers

INPUT:

```python
def validate_indian_mobile(mobile_number):
    # Convert to string to handle both string and integer inputs
    mobile_str = str(mobile_number)

    # Check if it contains exactly 10 digits
    if len(mobile_str) != 10:
        return False

    # Check if all characters are digits
    if not mobile_str.isdigit():
        return False

    # Check if it starts with 6, 7, 8, or 9
    if mobile_str[0] not in ['6', '7', '8', '9']:
        return False
    return True
if __name__ == "__main__":
    # Test cases
    test_numbers = [
        9876543210,      # Valid
        8123456789,      # Valid
        7234567890,      # Valid
        6345678901,      # Valid
        5123456789,      # Invalid - starts with 5
        9876543,         # Invalid - only 7 digits
        98765432101,     # Invalid - 11 digits
        "9876543210",    # Valid - as string
        "abcd567890",    # Invalid - contains letters
        "9876-543-210",  # Invalid - contains hyphens
    ]
    print("Indian Mobile Number Validation Results:")
    print("-" * 50)
    for number in test_numbers:
        result = validate_indian_mobile(number)
        status = "✓ Valid" if result else "✗ Invalid"
        print(f"{str(number):20} -> {status}")
```

OUTPUT:

```
Indian Mobile Number Validation Results:
--------------------------------------------------
9876543210          -> ✓ Valid
8123456789          -> ✓ Valid
7234567890          -> ✓ Valid
6345678901          -> ✓ Valid
5123456789          -> ✗ Invalid
9876543             -> ✗ Invalid
98765432101         -> ✗ Invalid
9876543210          -> ✓ Valid
abcd567890          -> ✗ Invalid
9876-543-210        -> ✗ Invalid
```

Task Description #6 (Loops – Armstrong Numbers in a Range)

Task: Write a function using AI that finds all Armstrong numbers in a user-specified range (e.g., 1 to 1000).

PROMPT:

```
generate python code to create function that finds Armstrong numbers
from 1 to 1000 use for loop and digit power logic and optimize it by
using list comprehensions
```

INPUT:

```python
def find_armstrong_for_loop(start=1, end=1000):
    armstrong_numbers = []
    for num in range(start, end + 1):
        digits = str(num)
        num_digits = len(digits)
        digit_sum = 0
        for digit in digits:
            digit_sum += int(digit) ** num_digits
        if digit_sum == num:
            armstrong_numbers.append(num)
    return armstrong_numbers

def find_armstrong_list_comprehension(start=1, end=1000):
    return [
        num for num in range(start, end + 1)
        if num == sum(int(digit) ** len(str(num)) for digit in str(num))
    ]

def find_armstrong_optimized(start=1, end=1000):
    return [
        num for num in range(start, end + 1)
        if sum(int(d) ** len(str(num)) for d in str(num)) == num
    ]

if __name__ == "__main__":
    print("Finding Armstrong Numbers from 1 to 1000")
    print("=" * 60)
    print("\nMethod 1: Using For Loop")
    print("-" * 60)
    armstrong_loop = find_armstrong_for_loop(1, 1000)
    print(f"Armstrong numbers found: {armstrong_loop}")
    print(f"Total count: {len(armstrong_loop)}")
    print("\nMethod 2: Using List Comprehension (Optimized)")
    print("-" * 60)
    armstrong_comp = find_armstrong_list_comprehension(1, 1000)
    print(f"Armstrong numbers found: {armstrong_comp}")
    print(f"Total count: {len(armstrong_comp)}")
    print("\nMethod 3: Optimized List Comprehension")
    print("-" * 60)
    armstrong_opt = find_armstrong_optimized(1, 1000)
    print(f"Armstrong numbers found: {armstrong_opt}")
    print(f"Total count: {len(armstrong_opt)}")
    print("\n" + "=" * 60)
    if armstrong_loop == armstrong_comp == armstrong_opt:
        print("✓ All methods produce identical results!")
    print("\nDetailed Breakdown:")
    print("-" * 60)
    for num in armstrong_opt:
        digits = str(num)
        power = len(digits)
        calculation = " + ".join([f"{d}^{power}" for d in digits])
        values = " + ".join([str(int(d) ** power) for d in digits])
        total = sum(int(d) ** power for d in digits)
        print(f"{num:4d} = {calculation:15s} = {values:15s} = {total}")
```

OUTPUT:

```
                > & C:/Python_codes/.venv/Scripts/python.exe c:/Python_codes/arms
Finding Armstrong Numbers from 1 to 1000
============================================================

Method 1: Using For Loop
------------------------------------------------------------
Armstrong numbers found: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Total count: 13

Method 2: Using List Comprehension (Optimized)
------------------------------------------------------------
Armstrong numbers found: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Total count: 13

Method 3: Optimized List Comprehension
------------------------------------------------------------
Armstrong numbers found: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Total count: 13

============================================================
✓ All methods produce identical results!

Detailed Breakdown:
------------------------------------------------------------
   1 = 1^1                = 1                = 1
   2 = 2^1                = 2                = 2
   3 = 3^1                = 3                = 3
   4 = 4^1                = 4                = 4
   5 = 5^1                = 5                = 5
   6 = 6^1                = 6                = 6
   7 = 7^1                = 7                = 7
   8 = 8^1                = 8                = 8
   9 = 9^1                = 9                = 9
 153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27    = 153
 370 = 3^3 + 7^3 + 0^3 = 27 + 343 + 0    = 370
 371 = 3^3 + 7^3 + 1^3 = 27 + 343 + 1    = 371
 407 = 4^3 + 0^3 + 7^3 = 64 + 0 + 343    = 407
```

Task Description #7 (Loops – Happy Numbers in a Range)

Task: Generate a function using AI that displays all Happy Numbers within a user-specified range (e.g., 1 to 500).

PROMPT:

generate python code to creatye a function that displays all happy numbers form 1 to 500 validate the numbers with known numbers and optimize it by usinga set to detect cycles instead of infinite loops

INPUT:

```python
def is_happy_number(n):
    seen = set()
    while n != 1 and n not in seen:
        seen.add(n)
        n = sum(int(digit) ** 2 for digit in str(n))
    return n == 1

def find_happy_numbers(start=1, end=500):
    return [num for num in range(start, end + 1) if is_happy_number(num)]

def display_happy_numbers(start=1, end=500):
    happy_nums = find_happy_numbers(start, end)
    print(f"Happy Numbers from {start} to {end}:")
    print("-" * 50)
    for i, num in enumerate(happy_nums, 1):
        print(f"{num:4d}", end="  ")
        if i % 10 == 0:
            print()
    print(f"\n\nTotal happy numbers: {len(happy_nums)}")
    return happy_nums

def is_happy_number_detailed(n):
    seen = set()
    original = n
    steps = []
    while n != 1 and n not in seen:
        seen.add(n)
        steps.append(n)
        n = sum(int(digit) ** 2 for digit in str(n))
    steps.append(n)
    return n == 1, steps

if __name__ == "__main__":
    happy_numbers = display_happy_numbers(1, 500)

    known_happy = [1, 7, 10, 13, 19, 23, 28, 31]
    known_unhappy = [2, 3, 4, 5, 6, 8, 9, 11]

    print("\n" + "=" * 50)
```

```python
if __name__ == "__main__":
    happy_numbers = display_happy_numbers(1, 500)

    known_happy = [1, 7, 10, 13, 19, 23, 28, 31]
    known_unhappy = [2, 3, 4, 5, 6, 8, 9, 11]

    print("\n" + "=" * 50)
    print("Validation with Known Numbers:")
    print("=" * 50)
    print("\nKnown Happy Numbers:")
    for num in known_happy:
        result = is_happy_number(num)
        status = "✓" if result else "✗"
        print(f"{status} {num}: {result}")

    print("\nKnown Unhappy Numbers:")
    for num in known_unhappy:
        result = is_happy_number(num)
        status = "✓" if not result else "✗"
        print(f"{status} {num}: {not result}")

    print("\n" + "=" * 50)
    print("Detailed Trace Examples:")
    print("=" * 50)
    test_cases = [7, 19, 2, 4]
    for num in test_cases:
        is_happy, steps = is_happy_number_detailed(num)
        status = "Happy" if is_happy else "Unhappy"
        print(f"\n{num} ({status}):")
        print(f"Steps: {' → '.join(map(str, steps))}")
```

OUTPUT:

```
Happy Numbers from 1 to 500:
--------------------------------------------------
   1     7    10    13    19    23    28    31    32    44
  49    68    70    79    82    86    91    94    97   100
 103   109   129   130   133   139   167   176   188   190
 192   193   203   208   219   226   230   236   239   262
 263   280   291   293   301   302   310   313   319   320
 326   329   331   338   356   362   365   367   368   376
 379   383   386   391   392   397   404   409   440   446
 464   469   478   487   490   496

Total happy numbers: 76

==================================================
Validation with Known Numbers:
==================================================

Known Happy Numbers:
✓ 1: True
✓ 7: True
✓ 10: True
✓ 13: True
✓ 19: True
✓ 23: True
✓ 28: True
✓ 31: True

Known Unhappy Numbers:
✓ 2: True
✓ 3: True
✓ 4: True
✓ 5: True
✓ 6: True
✓ 8: True
✓ 9: True
✓ 11: True

==================================================
Detailed Trace Examples:
==================================================

7 (Happy):
Steps: 7 → 49 → 97 → 130 → 10 → 1

19 (Happy):
Steps: 19 → 82 → 68 → 100 → 1

2 (Unhappy):
Steps: 2 → 4 → 16 → 37 → 58 → 89 → 145 → 42 → 20 → 4

4 (Unhappy):
Steps: 4 → 16 → 37 → 58 → 89 → 145 → 42 → 20 → 4
```

Task Description #8 (Loops – Strong Numbers in a Range)

PROMPT:

Generate a function to display all Strong Numbers within a given range using loops, and regenerate an optimized version using precomputed factorials.

INPUT:

```python
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

def precompute_factorials(max_digit=9):
    return {i: factorial(i) for i in range(max_digit + 1)}

def is_strong_number(n, factorial_dict):
    digit_sum = sum(factorial_dict[int(d)] for d in str(n))
    return digit_sum == n

def find_strong_numbers(start=1, end=100, factorial_dict=None):
    if factorial_dict is None:
        factorial_dict = precompute_factorials()
    return [num for num in range(start, end + 1) if is_strong_number(num, factorial_dict)]

def display_strong_numbers(start=1, end=100):
    factorial_dict = precompute_factorials()
    strong_nums = find_strong_numbers(start, end, factorial_dict)
    print(f"Strong Numbers from {start} to {end}:")
    for num in strong_nums:
        digits = [int(d) for d in str(num)]
        calc = " + ".join([f"{d}!" for d in digits])
        values = " + ".join([str(factorial_dict[d]) for d in digits])
        print(f"{num}: {calc} = {values} = {num}")
    print(f"Total: {len(strong_nums)}\n")
    return strong_nums

if __name__ == "__main__":
    print("=" * 60)
    print("STRONG NUMBERS FINDER")
    print("=" * 60)
    strong_nums = display_strong_numbers(1, 150)
    factorial_dict = precompute_factorials()
    print("Precomputed Factorials:", factorial_dict)
```

Output:

```
============================================================
STRONG NUMBERS FINDER
============================================================
Strong Numbers from 1 to 150:
1: 1! = 1 = 1
2: 2! = 2 = 2
145: 1! + 4! + 5! = 1 + 24 + 120 = 145
Total: 3

Precomputed Factorials: {0: 1, 1: 1, 2: 2, 3: 6, 4: 24, 5: 120, 6: 720, 7: 5040, 8: 40320, 9: 362880}
PS C:\Python_codes>
```

Task #9 – Few-Shot Prompting for Nested Dictionary Extraction

Objective

Use few-shot prompting (2–3 examples) to instruct the AI to create a function that parses a nested dictionary representing student information.

Requirements

• The function should extract and return:

o Full Name

o Branch

o SGPA

PROMPT:

```
. Create a Python function that extracts full name, branch, and SGPA
from a nested dictionary representing student information.
```

INPUT:

```python
def extract_student_info(student_dict):
    full_name = student_dict.get('name', {}).get('first', '') + ' ' + student_dict.get('name', {}).get('last', '')
    branch = student_dict.get('academic', {}).get('branch', 'N/A')
    sgpa = student_dict.get('academic', {}).get('sgpa', 'N/A')
    return full_name.strip(), branch, sgpa
def extract_multiple_students(students_list):
    return [extract_student_info(student) for student in students_list]
def display_student_info(student_dict):
    full_name, branch, sgpa = extract_student_info(student_dict)
    print(f"Full Name: {full_name}")
    print(f"Branch: {branch}")
    print(f"SGPA: {sgpa}")
def display_all_students(students_list):
    print("=" * 60)
    print("STUDENT INFORMATION")
    print("=" * 60)
    for i, student in enumerate(students_list, 1):
        print(f"\nStudent {i}:")
        print("-" * 30)
        display_student_info(student)
if __name__ == "__main__":
    students = [
        {
            'name': {'first': 'John', 'last': 'Smith'},
            'academic': {'branch': 'Computer Science', 'sgpa': 8.5}
        },
        {
            'name': {'first': 'Sarah', 'last': 'Johnson'},
            'academic': {'branch': 'Electronics', 'sgpa': 8.2}
        },
        {
            'name': {'first': 'Mike', 'last': 'Davis'},
            'academic': {'branch': 'Mechanical', 'sgpa': 7.9}
        },
        {
            'name': {'first': 'Emily', 'last': 'Wilson'},
            'academic': {'branch': 'Civil', 'sgpa': 8.7}
        }
    ]
    display_all_students(students)
    print("\n" + "=" * 60)
    print("EXTRACTED DATA:")
    print("=" * 60)
    extracted = extract_multiple_students(students)
    for i, (name, branch, sgpa) in enumerate(extracted, 1):
        print(f"{i}. {name} | {branch} | SGPA: {sgpa}")
```

Task Description #10 (Loops – Perfect Numbers in a Range)

Task: Generate a function using AI that displays all Perfect Numbers within a user-specified range (e.g., 1 to 1000).

PROMPT:

Generate a Python function that prints all Perfect Numbers in a user given range, and regenerate an optimized version by checking divisors only up to square root of the number.

INPUT:

```python
def find_divisors_standard(n):
    divisors = []
    for i in range(1, n):
        if n % i == 0:
            divisors.append(i)
    return divisors

def is_perfect_number_standard(n):
    return sum(find_divisors_standard(n)) == n

def find_perfect_numbers_standard(start, end):
    return [num for num in range(start, end + 1) if is_perfect_number_standard(num)]

def find_divisors_optimized(n):
    divisors = [1]
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            divisors.append(i)
            if i != n // i:
                divisors.append(n // i)
    return divisors

def is_perfect_number_optimized(n):
    if n <= 1:
        return False
    return sum(find_divisors_optimized(n)) == n

def find_perfect_numbers_optimized(start, end):
    return [num for num in range(start, end + 1) if is_perfect_number_optimized(num)]

def display_perfect_numbers_standard(start, end):
    print("Perfect Numbers (Standard Method):")
    print("-" * 50)
    perfect = find_perfect_numbers_standard(start, end)
    for num in perfect:
        divisors = find_divisors_standard(num)
        print(f"{num}: {' + '.join(map(str, divisors))} = {sum(divisors)}")
    print(f"Total found: {len(perfect)}\n")
    return perfect
```

```python
def find_perfect_numbers_optimized(start, end):
    return [num for num in range(start, end + 1) if is_perfect_number_optimized(num)]

def display_perfect_numbers_standard(start, end):
    print("Perfect Numbers (Standard Method):")
    print("-" * 50)
    perfect = find_perfect_numbers_standard(start, end)
    for num in perfect:
        divisors = find_divisors_standard(num)
        print(f"{num}: {' + '.join(map(str, divisors))} = {sum(divisors)}")
    print(f"Total found: {len(perfect)}\n")
    return perfect

def display_perfect_numbers_optimized(start, end):
    print("Perfect Numbers (Optimized - Square Root Method):")
    print("-" * 50)
    perfect = find_perfect_numbers_optimized(start, end)
    for num in perfect:
        divisors = sorted(find_divisors_optimized(num))
        print(f"{num}: {' + '.join(map(str, divisors))} = {sum(divisors)}")
    print(f"Total found: {len(perfect)}\n")
    return perfect

if __name__ == "__main__":
    start = int(input("Enter start of range: "))
    end = int(input("Enter end of range: "))
    print("\n" + "=" * 60)
    print("PERFECT NUMBERS FINDER")
    print("=" * 60 + "\n")
    standard = display_perfect_numbers_standard(start, end)
    print("=" * 60 + "\n")
    optimized = display_perfect_numbers_optimized(start, end)
    print("=" * 60)
    print(f"Both methods match: {standard == optimized}\n")
```

OUTPUT:

```
Enter start of range: 1
Enter end of range: 100

============================================================
PERFECT NUMBERS FINDER
============================================================

Perfect Numbers (Standard Method):
--------------------------------------------------
6: 1 + 2 + 3 = 6
28: 1 + 2 + 4 + 7 + 14 = 28
Total found: 2

============================================================

Perfect Numbers (Optimized - Square Root Method):
--------------------------------------------------
6: 1 + 2 + 3 = 6
28: 1 + 2 + 4 + 7 + 14 = 28
Total found: 2

============================================================
Both methods match: True
```