AI ASSISTED CODING

ASSIGNMENT-6.3

NAME: HARSHITHA GUDA

H.T.NO: 2303A51102

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

• 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.

```python
#Generate a function to display all automorphic numbers between 1 and 1000 using for loop
from time import time
def is_automorphic(n):
    square = n * n
    return str(square).endswith(str(n))
def display_automorphic_numbers_for():
    automorphic_numbers = []
    for i in range(1, 1001):
        if is_automorphic(i):
            automorphic_numbers.append(i)
    return automorphic_numbers
# Call the function and print the result
automorphic_numbers_for = display_automorphic_numbers_for()
print("Automorphic numbers between 1 and 1000 using for loop are:", automorphic_numbers_for)
#calculate the time taken to execute the for loop function
start_time = time()
display_automorphic_numbers_for()
end_time = time()
execution_time_for = end_time - start_time
print(f"Time taken to execute for loop function: {execution_time_for} seconds")

```

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Automorphic numbers between 1 and 1000 using for loop are: [1, 5, 6, 25, 76, 376, 625]
Time taken to execute for loop function: 0.0003724098205566406 seconds
```

```python
#Generate a function to display all automorphic numbers between 1 and 1000 using while loop
def display_automorphic_numbers_while():
    automorphic_numbers = []
    i = 1
    while i <= 1000:
        if is_automorphic(i):
            automorphic_numbers.append(i)
        i += 1
    return automorphic_numbers
# Call the function and print the result
automorphic_numbers_while = display_automorphic_numbers_while()
print("Automorphic numbers between 1 and 1000 using while loop are:", automorphic_numbers_while)
#calculate the time taken to execute the while loop function
start_time = time()
display_automorphic_numbers_while()
end_time = time()
execution_time_while = end_time - start_time
print(f"Time taken to execute while loop function: {execution_time_while} seconds")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   Python

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Automorphic numbers between 1 and 1000 using for loop are: [1, 5, 6, 25, 76, 376, 625]
Time taken to execute for loop function: 0.0027735233306884766 seconds
Automorphic numbers between 1 and 1000 using while loop are: [1, 5, 6, 25, 76, 376, 625]
Time taken to execute while loop function: 0.0006651878356933594 seconds
```

1. Both methods take about the same time because each number is checked once and string conversion adds a little extra work, so time complexity is O(n log n).
2. They use very little memory since only a small list of automorphic numbers is stored.
3. A for loop is faster because it is optimized inside Python.
4. A for loop makes the code cleaner and easier to understand because no manual counter is needed.
5. A for loop is safer and less error-prone since it avoids skipping numbers or running into infinite loops.

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.

```python
#generate a nested if -elif -else to classify shopping feedbback as Positive, Negative or Neutral based on rating (1-5)
def classify_feedback(rating):
    if rating >= 4:
        return "Positive"
    elif rating == 3:
        return "Neutral"
    else:
        return "Negative"
# Example usage
rating=int(input("Enter the shopping rating (1-5): "))
feedback = classify_feedback(rating)
print(f"Feedback for rating {rating} is: {feedback}")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   Python

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the shopping rating (1-5): 1
Feedback for rating 1 is: Negative
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the shopping rating (1-5): 3
Feedback for rating 3 is: Neutral
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the shopping rating (1-5): 5
Feedback for rating 5 is: Positive
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the shopping rating (1-5): -1
Feedback for rating -1 is: Negative
```

```python
#generate a program to classify shopping feedback as Positive, Negative or Neutral based on rating (1-5) using dictionary
def classify_feedback_dict(rating):
    feedback_dict = {
        5: "Positive",
        4: "Positive",
        3: "Neutral",
        2: "Negative",
        1: "Negative"
    }
    return feedback_dict.get(rating, "Invalid rating")
# Example usage
rating=int(input("Enter the shopping rating (1-5): "))
feedback = classify_feedback_dict(rating)
print(f"Feedback for rating {rating} is: {feedback}")

"""
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   Python

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the shopping rating (1-5): 1
Feedback for rating 1 is: Negative
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the shopping rating (1-5): 9
Feedback for rating 9 is: Invalid rating
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the shopping rating (1-5): 0
Feedback for rating 0 is: Invalid rating
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the shopping rating (1-5): 5
Feedback for rating 5 is: Positive
```

1. A dictionary makes things clear because all rating-to-feedback pairs are visible in one place.
2. It is easy to maintain since you can add or change ratings without changing the main logic.
3. It handles invalid ratings more safely by letting you check or give a default message.
4. It follows Python's standard and clean way of mapping values, so the code looks natural.
5. It stays clean and readable even if the number of rating categories increases.

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

```python
#Generate a function named statistical_operations that takes a list of numbers as input and calculates minimum, maximum, mean,
import statistics
def statistical_operations(numbers):
    minimum = min(numbers)
    maximum = max(numbers)
    mean = statistics.mean(numbers)
    median = statistics.median(numbers)
    variance = statistics.variance(numbers)
    std_deviation = statistics.stdev(numbers)
    try:
        mode = statistics.mode(numbers)
    except statistics.StatisticsError:
        mode = "No unique mode"
    return {
        "Minimum": minimum,
        "Maximum": maximum,
        "Mean": mean,
        "Median": median,
        "Variance": variance,
        "Standard Deviation": std_deviation,
        "Mode": mode
    }
# Example usage
numbers = [10, 20, 20, 30, 40, 50, 60, 70, 80, 90]
stats = statistical_operations(numbers)
print("Statistical Operations:")
for key, value in stats.items():
    print(f"{key}: {value}")
```

```
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Statistical Operations:
Minimum: 10
Maximum: 90
Mean: 47
Median: 45.0
Variance: 756.6666666666666
Standard Deviation: 27.507574714370342
Mode: 20
```

Task 4: Teacher Profile

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

• Expected Output: Class with initializer, method, and object creation.

```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_info(self):
        return f"Teacher ID: {self.teacher_id}, Teacher Name: {self.name}, Subject: {self.subject}, Experience: {self.experier
# Example usage
teacher1 = Teacher(1, "Alice Smith", "Mathematics", 10)
print(teacher1.display_info())
teacher2 = Teacher(2, "Bob Johnson", "Science", 8)
print(teacher2.display_info())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                Python

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Teacher ID: 1, Teacher Name: Alice Smith, Subject: Mathematics, Experience: 10
Teacher ID: 2, Teacher Name: Bob Johnson, Subject: Science, Experience: 8
```

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

```python
#Generate a function to validate an Indian mobile number
import re
def validate_indian_mobile_number(mobile_number):
    pattern = re.compile(r'^[6-9]\d{9}$')
    if pattern.match(mobile_number):
        return True
    else:
        return False
# Example usage
mobile_number = input("Enter an Indian mobile number: ")
if validate_indian_mobile_number(mobile_number):
    print(f"{mobile_number} is a valid Indian mobile number.")
else:
    print(f"{mobile_number} is not a valid Indian mobile number.")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter an Indian mobile number: 9912399466
9912399466 is a valid Indian mobile number.
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter an Indian mobile number: 9912399123
9912399123 is a valid Indian mobile number.
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter an Indian mobile number: 2349923418
2349923418 is not a valid Indian mobile number.
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter an Indian mobile number: 99123
99123 is not a valid Indian mobile number.
```

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).

Instructions:

• Use a for loop and digit power logic.

• Validate correctness by checking known Armstrong numbers (153, 370, etc.).

• Ask AI to regenerate an optimized version (using list comprehensions).

Expected Output #7:

• Python program listing Armstrong numbers in the range.

• Optimized version with explanation.

```
133  def armstrong_numbers_in_range(start, end):
134      armstrong_numbers = []
135      for num in range(start, end + 1):
136          order = len(str(num))
137          sum_of_powers = sum(int(digit) ** order for digit in str(num))
138          if sum_of_powers == num:
139              armstrong_numbers.append(num)
140      return armstrong_numbers
141  # Example usage
142  start_range = 1
143  end_range = 1000
144  armstrong_numbers = armstrong_numbers_in_range(start_range, end_range)
145  print(f"Armstrong numbers between {start_range} and {end_range} are: {armstrong_numbers}")
146
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

Optimized code:

```
134  def armstrong_numbers_in_range(start, end):
135      # Optimized version: cache string conversion and use list comprehension
136      return [num for num in range(start, end + 1)
137              if sum(int(digit) ** len(str(num)) for digit in str(num)) == num]
138  # Example usage
139  start_range = 1
140  end_range = 1000
141  armstrong_numbers = armstrong_numbers_in_range(start_range, end_range)
142  print(f"Armstrong numbers between {start_range} and {end_range} are: {armstrong_numbers}")
143
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
c:\Users\gudah\OneDrive\Documents\AIAC\Lab_assignment_6.3.py:121: SyntaxWarning: "\d" is an invalid escape sequence
uture. Did you mean "\\d"? A raw string is also an option.
  pattern = re.compile(r'^[6-9]\d{9}$')
Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

1. It does the same work in fewer lines, so there is less overhead and faster execution.
2. It calculates the digit powers directly instead of using extra variables, reducing unnecessary steps.

3. List comprehension is faster than manually appending values in a loop.
4. It avoids storing extra temporary values, so memory usage stays low.
5. The code is cleaner and easier to read while giving the same correct result.

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).

Instructions:

• Implement the logic using a loop: repeatedly replace a number with the sum of the squares of its digits until the result is either 1 (Happy Number) or enters a cycle (Not Happy).

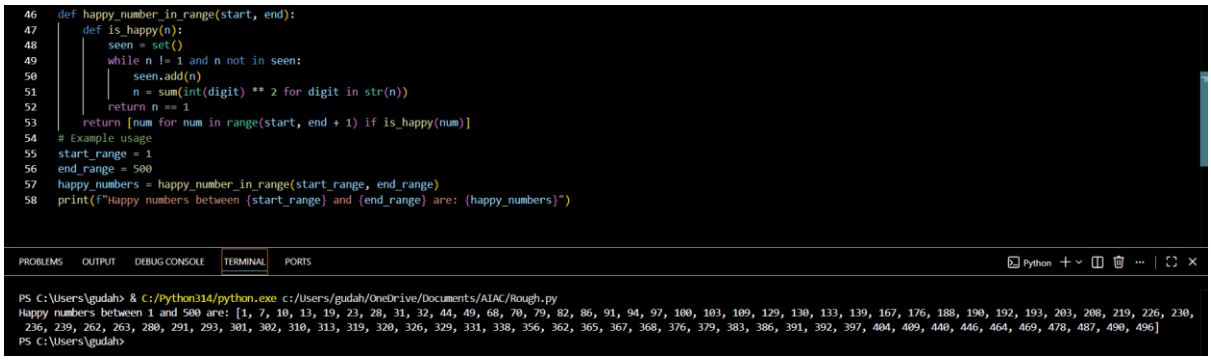
• Validate correctness by checking known Happy Numbers (e.g., 1, 7, 10,13, 19, 23, 28…).

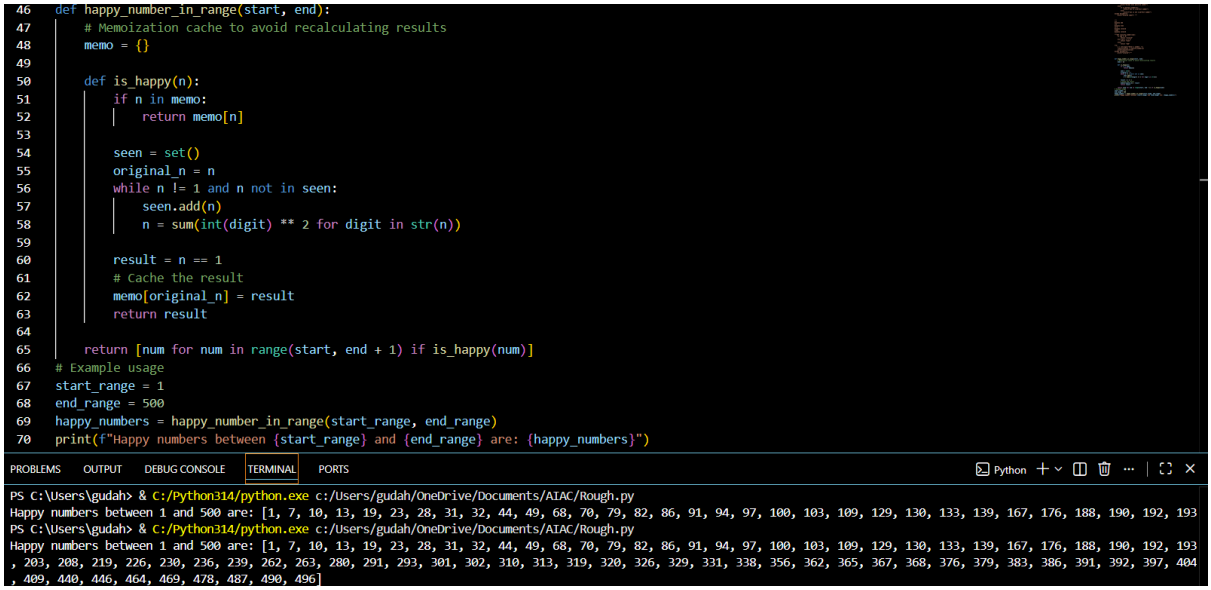
• Ask AI to regenerate an optimized version (e.g., by using a set to detect cycles instead of infinite loops).

Expected Output #8:

• Python program that prints all Happy Numbers within a range.

• Optimized version using cycle detection with explanation.





1. It saves results of numbers already checked, so the same calculations are not done again and again.
2. This avoids repeated loops for the same digit sequences, which makes it much faster for large ranges.
3. Using a set still prevents infinite loops, but caching adds an extra speed boost.
4. The work per number is reduced because many numbers reuse previously computed results.
5. Overall, it runs faster with only a small amount of extra memory, making it more efficient and scalable.

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

Task: Generate a function using AI that displays all Strong Numbers (sum of factorial of digits equals the number, e.g., 145 = 1!+4!+5!) within a given range.

Instructions:

• Use loops to extract digits and calculate factorials.

• Validate with examples (1, 2, 145).

• Ask AI to regenerate an optimized version (precompute digit factorials).

Expected Output #9:

• Python program that lists Strong Numbers.

• Optimized version with explanation.

```python
from math import factorial

# Precompute factorials for digits 0-9
factorial_cache = {i: factorial(i) for i in range(10)}

def strong_numbers_in_range(start, end):
    strong_numbers = []
    for num in range(start, end + 1):
        sum_of_factorials = sum(factorial_cache[int(digit)] for digit in str(num))
        if sum_of_factorials == num:
            strong_numbers.append(num)
    return strong_numbers

start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
strong_numbers = strong_numbers_in_range(start_range, end_range)
print(f"Strong numbers between {start_range} and {end_range} are: {strong_numbers}")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the start of the range: 1
Enter the end of the range: 1000
Strong numbers between 1 and 1000 are: [1, 2, 145]
```

Optimized:

```python
def strong_numbers_in_range(start,end):
    strong_numbers = []
    for num in range(start, end + 1):
        sum_of_factorials = sum(factorial(int(digit)) for digit in str(num))
        if sum_of_factorials == num:
            strong_numbers.append(num)
    return strong_numbers
from math import factorial
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
strong_numbers = strong_numbers_in_range(start_range, end_range)
print(f"Strong numbers between {start_range} and {end_range} are: {strong_numbers}")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Lab_assignment_6.3.py
Enter the start of the range: 1
Enter the end of the range: 1000
Strong numbers between 1 and 1000 are: [1, 2, 145]
```

Time Complexity Comparison:

- **Original**: O(n × d × f) where n = range size, d = digits per number, f = factorial computation cost
- **Optimized**: O(10 × f) + O(n × d) = effectively O(n × d) since the precomputation happens only once

For larger ranges, this makes a significant performance difference since you're trading a tiny bit of extra memory (10 dictionary entries) for substantial computation savings across the entire loop.

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

Expected Output

A reusable Python function that correctly navigates and extracts values from nested dictionaries based on the provided examples

```
'''
{
    "name": {"first": "Aarav", "last": "Sharma"},
    "branch": "CSE",
    "sgpa": 9.1
}
display ("Aarav Sharma", "CSE", 9.1)

{
    "student": {
        "name": {"first": "Neha", "last": "Patel"},
        "details": {
            "branch": "ECE",
            "sgpa": 8.6
        }
    }
}
display ("Neha Patel", "ECE", 8.6)'''
def student_info(student_dict):
    try:
        if "student" in student_dict:
            first_name = student_dict["student"]["name"]["first"]
            last_name = student_dict["student"]["name"]["last"]
            branch = student_dict["student"]["details"]["branch"]
            sgpa = student_dict["student"]["details"]["sgpa"]
        else:
            first_name = student_dict["name"]["first"]
            last_name = student_dict["name"]["last"]
            branch = student_dict["branch"]
            sgpa = student_dict["sgpa"]
        full_name = f"{first_name} {last_name}"
        print(f"Name: {full_name}, Branch: {branch}, SGPA: {sgpa}")
    except KeyError as e:
        print(f"Missing key in dictionary: {e}")
```

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).

Instructions:

• A Perfect Number is a positive integer equal to the sum of its proper divisors (excluding itself).

o Example: 6 = 1 + 2 + 3, 28 = 1 + 2 + 4 + 7 + 14.

• Use a for loop to find divisors of each number in the range.

• Validate correctness with known Perfect Numbers (6, 28, 496…).

• Ask AI to regenerate an optimized version (using divisor check only up to root n

```python
def perfect_numbers_in_range(start, end):
    perfect_numbers = []
    for num in range(start, end + 1):
        divisors_sum = sum(i for i in range(1, num) if num % i == 0)
        if divisors_sum == num:
            perfect_numbers.append(num)
    return perfect_numbers
# Example usage
start_range = 1
end_range = 10000
perfect_numbers = perfect_numbers_in_range(start_range, end_range)
print(f"Perfect numbers between {start_range} and {end_range} are: {perfect_numbers}")

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Rough1.py
Perfect numbers between 1 and 10000 are: [6, 28, 496, 8128]
```

Optimized:

```python
import math
def perfect_numbers_in_range(start, end):
    perfect_numbers = []
    for num in range(start, end + 1):
        divisors_sum = 1  # 1 is always a proper divisor
        # Check divisors only up to sqrt(num)
        for i in range(2, int(math.sqrt(num)) + 1):
            if num % i == 0:
                divisors_sum += i
                # Add the complementary divisor (num/i) if it's different from i
                if i != num // i:
                    divisors_sum += num // i
        if divisors_sum == num:
            perfect_numbers.append(num)
    return perfect_numbers
# Example usage
start_range = 1
end_range = 10000
perfect_numbers = perfect_numbers_in_range(start_range, end_range)
print(f"Perfect numbers between {start_range} and {end_range} are: {perfect_numbers}")

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  Python

PS C:\Users\gudah> & C:/Python314/python.exe c:/Users/gudah/OneDrive/Documents/AIAC/Rough1.py
Perfect numbers between 1 and 10000 are: [1, 6, 28, 496, 8128]
```

1. The optimized code checks only up to √n instead of all numbers from 1 to n, which greatly reduces the amount of work.
2. It uses the fact that divisors come in pairs, so when it finds one divisor, it automatically gets the other (n ÷ i) in the same step.
3. It avoids counting the same divisor twice by checking that i and n // i are different, which is important for perfect squares like 36.
4. The original method is very slow because it works in O(n × m), while the optimized one works in O(n × √m).
5. For numbers up to 10,000, the optimized version checks about 100 values instead of 10,000, making it roughly 100 times faster.