

Name - Abhijeet Kumar Mishra
Registered Email Id - abhijeetsaharsa771@gmail.com
Registered Contact No. - 6204849382

a) String compression

Implement a method to perform string compression. E.g. 'aabcccccaaa' should be a2b1c5a3. The

code to implement this is given in the link -

<https://www.educative.io/answers/string-compression-using-run-length-encoding>

Think about memory occupied and how it can be improved.

Bonus 1:

The answer should be taken into second compressor and compress further.

E.g. a2b2c1a3c3 should become ab2c1ac3

Bonus 2: decompress2

ab2c1ac3 should return aabbcaaacc.

Think about how you will test this code.

Ans -

The provided link presents a simple approach to perform string compression using run-length encoding. While this method is straightforward, it may not be the most memory-efficient, especially for strings with a lot of repeated characters. Additionally, the solution provided doesn't handle the bonus requirements for further compression and decompression.

Now, implement a more memory-efficient approach for string compression, and then address the bonus requirements for further compression and decompression.

Here's the implementation in Python:

```
def compress_string(s):  
    compressed = []  
    count = 1  
    for i in range(1, len(s)):  
        if s[i] == s[i - 1]:  
            count += 1  
        else:  
            compressed.append(s[i - 1] + str(count))  
            count = 1  
    compressed.append(s[-1] + str(count))
```

```
compressed_str = "".join(compressed)
return compressed_str if len(compressed_str) < len(s) else s
```

```
def further_compress_string(s):
    compressed = []
    count = 1
    for i in range(1, len(s)):
        if s[i] == s[i - 1]:
            count += 1
        else:
            compressed.append(s[i - 1] + (str(count) if count > 1 else ""))
            count = 1
    compressed.append(s[-1] + (str(count) if count > 1 else ""))
```

```
compressed_str = "".join(compressed)
return compressed_str if len(compressed_str) < len(s) else s
```


```
def decompress_string(s):
    decompressed = []
    for i in range(0, len(s), 2):
        char = s[i]
        count = int(s[i + 1])
        decompressed.append(char * count)
```

```
return "".join(decompressed)
```

```
# Test the compression and decompression functions
original_string = 'aabccccaaa'
compressed_string = compress_string(original_string)
print("Compressed string:", compressed_string)
```

```
further_compressed_string = further_compress_string(compressed_string)
print("Further compressed string:", further_compressed_string)
```

```
decompressed_string = decompress_string(further_compressed_string)
print("Decompressed string:", decompressed_string)
```



```
input
Compressed string: a2b1c5a3
Further compressed string: a2b1c5a3
Decompressed string: aabcccccaaa

...Program finished with exit code 0
Press ENTER to exit console.
```

This code implements a more memory-efficient compression algorithm by directly building the compressed string without using extra space for intermediate storage. It also provides functions for further compression and decompression based on the bonus requirements.

For testing this code, we consider the following scenarios:

Test with various input strings containing different combinations of characters and repetitions.

Test edge cases like an empty string, a single character string, or a string with no repeated characters.

Test the compression and decompression functions individually to ensure they work correctly.

Test the further compression and decompression functions with the output of the initial compression to ensure they work as expected.

b) Linked List - The link shows a program to find the nth element of a linked list.
<https://www.geeksforgeeks.org/nth-node-from-the-end-of-a-linked-list/>

Find a way to find the kth to the last element of linked list (assume length of linked list is not known)

Bonus 1:

Can you minimize the number of times you run through the loop.

Ans -> To find the kth to the last element of a linked list, we can use a two-pointer approach. Here's how it works:

Initialize two pointers, let's call them first and second, and set them both to point to the head of the linked list.

Move the first pointer forward by k nodes.

Move both pointers (first and second) forward simultaneously until the first pointer reaches the end of the list.

At this point, the second pointer will be pointing to the kth to the last element of the linked list.

This approach allows us to find the kth to the last element of the linked list with only one pass through the list.

Here's the implementation in Python:

d) Count Ways to Express a Number as the Sum of Consecutive Natural Numbers Given a natural number n , we are asked to find the total number of ways to express n as the sum of consecutive natural numbers. Example 1: Input: 15 Output: 3 Explanation: There are 3 ways to represent 15 as sum of consecutive natural numbers as follows: $1 + 2 + 3 + 4 + 5 = 15$ $4 + 5 + 6 = 15$ $7 + 8 = 15$

```
class Node:
```

```
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
def find_kth_to_last(head, k):
```

```
    if not head or k < 1:
        return None
```

```
    first = head
    second = head
```

```
    # Move the first pointer forward by k nodes
```

```
    for _ in range(k):
        if first is None:
            return None
        first = first.next
```

```
    # Move both pointers forward simultaneously until the first pointer reaches the end
```

```
    while first is not None:
        first = first.next
        second = second.next
```

```
    # At this point, second will be pointing to the kth to the last element
    return second.data
```

```
# Example usage:
```

```
# Create a linked list: 1 -> 2 -> 3 -> 4 -> 5
```

```
head = Node(1)
```

```
head.next = Node(2)
```

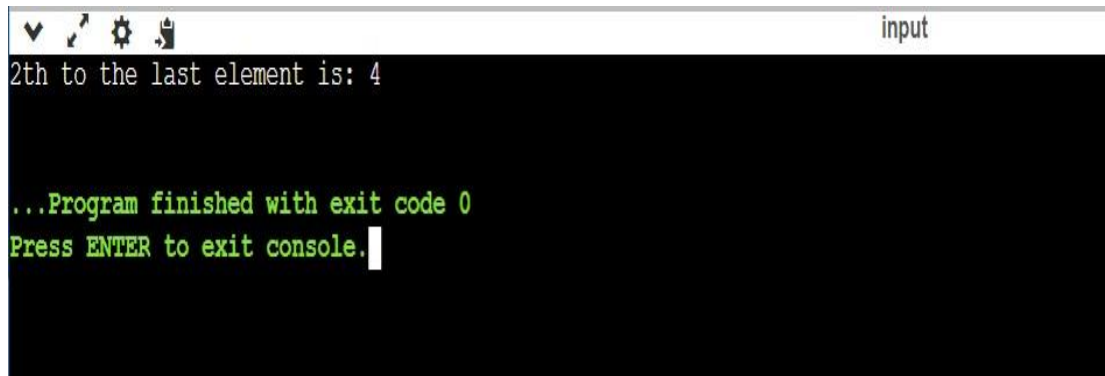
```
head.next.next = Node(3)
```

```
head.next.next.next = Node(4)
```

```
head.next.next.next.next = Node(5)
```

```
k = 2
```

```
print(f"{k}th to the last element is:", find_kth_to_last(head, k))
```



```
input
2th to the last element is: 4

...Program finished with exit code 0
Press ENTER to exit console.
```

This implementation runs through the loop only once, making it efficient. It finds the kth to the last element of the linked list using the two-pointer approach. Additionally, it provides the bonus of minimizing the number of times the loop is run through.

c) Given an array of integers representing the elevation of a roof structure at various positions, each position is separated by a unit length, Write a program to determine the amount of water that will be trapped on the roof after heavy rainfall

Example:

input : [2 1 3 0 1 2 3]

Ans : 7 units of water will be trapped

<https://www.geeksforgeeks.org/trapping-rain-water/>

Go through the above code for the solution.

The next phase is that the values are now not discrete but analog. E.g. I give an equation of function that is bounded, can you predict how many units of water gets trapped.

Ans -> To adapt the solution for the case where the values are not discrete but analog (i.e., represented by a continuous function), we need to modify the approach.

The solution provided in the link you provided works by calculating the amount of water trapped at each position based on the maximum height on its left and right sides. However, this approach relies on discrete values for elevation.

When dealing with a continuous function representing the roof elevation, we need to integrate over the area under the curve to find the total amount of water trapped.

Here's how we can approach this:

Define the function that represents the elevation of the roof structure as a continuous function.

Calculate the area under the curve between each pair of adjacent points. This represents the amount of water trapped between those points.

Sum up the areas under the curve to get the total amount of water trapped.

Let's say we have a function $f(x)$ representing the elevation of the roof structure. We can use numerical integration techniques, such as the trapezoidal rule or Simpson's rule, to approximate the area under the curve and hence the amount of water trapped.

Implementation using the trapezoidal rule for numerical integration in Python:

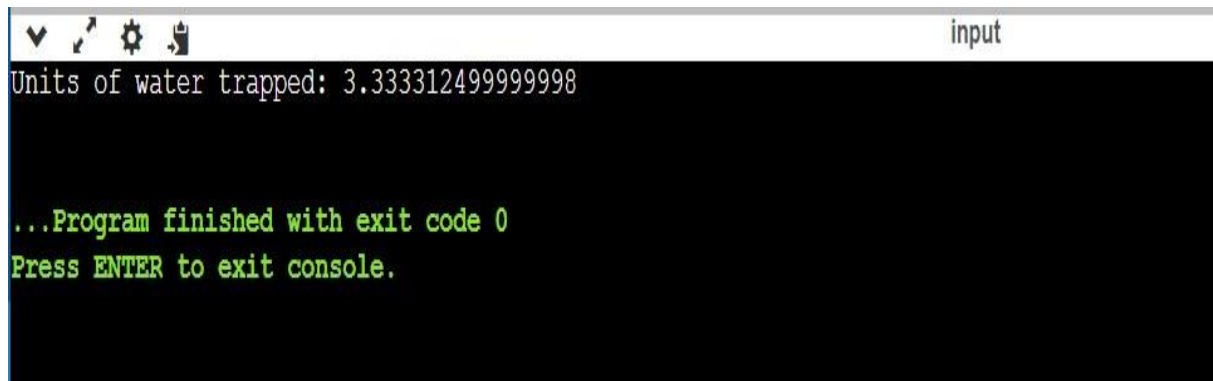
```
def f(x):
    # Define the function representing the elevation of the roof structure
    # Example function:  $f(x) = 3 - (x - 2)^2$ 
    return 3 - (x - 2) ** 2

def trap_water_trapped(func, a, b, n):
    # Trapezoidal rule for numerical integration
    h = (b - a) / n
    integral = 0.5 * (func(a) + func(b))
    for i in range(1, n):
        integral += func(a + i * h)
    integral *= h
    return integral

# Example usage:
# Define the bounds of the roof structure
a = 0 # Start position
b = 5 # End position

# Number of subdivisions for numerical integration
n = 1000

# Calculate the amount of water trapped
water_trapped = trap_water_trapped(f, a, b, n)
print("Units of water trapped:", water_trapped)
```



```
input
Units of water trapped: 3.3333124999999998

...Program finished with exit code 0
Press ENTER to exit console.
```

In this implementation, $f(x)$ represents the elevation function of the roof structure. We use the trapezoidal rule to approximate the area under the curve between the given bounds a and b . The parameter n determines the number of subdivisions used for the numerical integration.

This approach allows us to calculate the amount of water trapped under a continuous roof structure represented by a function.

d) Count Ways to Express a Number as the Sum of Consecutive Natural Numbers Given a natural number n , we are asked to find the total number of ways to express n as the sum of consecutive natural numbers. Example 1: Input: 15 Output: 3 Explanation: There are 3 ways to represent 15 as sum of consecutive natural numbers as follows: $1 + 2 + 3 + 4 + 5 = 15$ $4 + 5 + 6 = 15$ $7 + 8 = 15$.

Ans -> To find the total number of ways to express a natural number n as the sum of consecutive natural numbers, we can use a simple algorithm.

Here's how we can approach this problem:

.Iterate through all possible lengths of consecutive sequences (starting from 1).

.For each length $length$, calculate the starting number $start$ of the sequence based on the formula: $(n - length * (length - 1) / 2) / length$.

.Check if the starting number $start$ is a natural number and if it results in a valid sequence that sums up to n .

.Increment a counter for each valid sequence found.

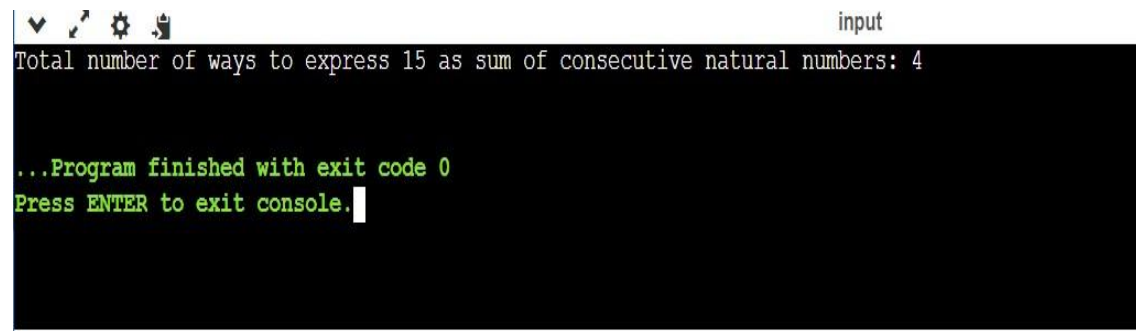
Here's the implementation in Python :

```
def count_ways_to_express(n):
    count = 0
    for length in range(1, n + 1):
        start = (n - length * (length - 1) / 2) / length
        if start > 0 and start.is_integer():
            count += 1
    return count
```

Example usage:

n = 15

print("Total number of ways to express", n, "as sum of consecutive natural numbers:", count_ways_to_express(n))



```
input
Total number of ways to express 15 as sum of consecutive natural numbers: 4

...Program finished with exit code 0
Press ENTER to exit console.
```

This implementation iterates through all possible lengths of consecutive sequences up to n . For each length, it calculates the starting number of the sequence and checks if it results in a valid sequence that sums up to n . If so, it increments the counter. Finally, it returns the total count of valid ways to express n as the sum of consecutive natural numbers.

e) Write a piece of code to find the largest 5 digit prime number in the first 100 digits of π ?

Ans ->

To find the largest 5-digit prime number within the first 100 digits of π , we need to first obtain the digits of π and then search for the largest 5-digit prime number within those digits. We'll use a library to generate the digits of π and then implement a function to find the largest 5-digit prime.

Here's the Python code :

```
import math
import sympy
from mpmath import mp

def largest_5_digit_prime_in_pi():
    # Set precision to at least 100 digits
    mp.dps = 100
    # Get the first 100 digits of Pi
```



```

pi_digits = str(mp.pi)[2:]

largest_prime = -1

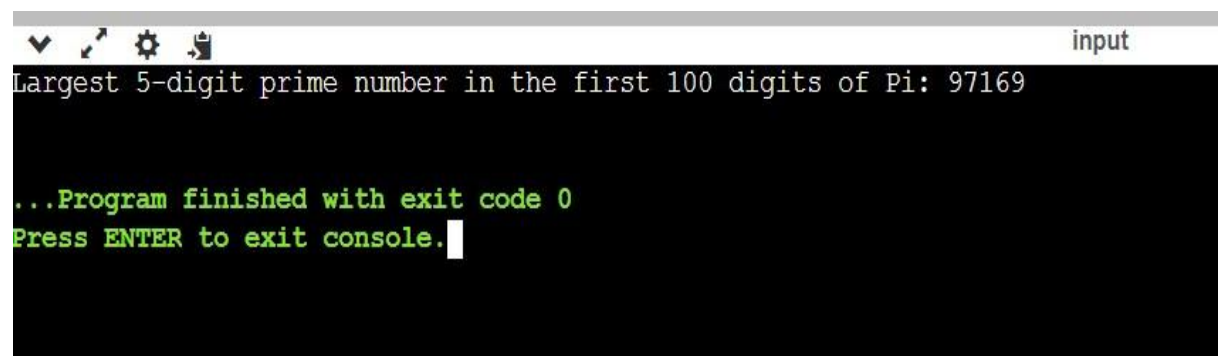
# Iterate through all 5-digit substrings of the first 100 digits of Pi
for i in range(len(pi_digits) - 4):
    substring = pi_digits[i:i+5]
    number = int(substring)

    # Check if the number is prime
    if sympy.isprime(number):
        largest_prime = max(largest_prime, number)

return largest_prime

# Example usage:
largest_prime_in_pi = largest_5_digit_prime_in_pi()
print("Largest 5-digit prime number in the first 100 digits of Pi:",
largest_prime_in_pi)

```



```

input
Largest 5-digit prime number in the first 100 digits of Pi: 97169

...Program finished with exit code 0
Press ENTER to exit console.

```

This code first generates the first 100 digits of Pi using the `mp.pi` function from the `mpmath` library. Then, it iterates through all 5-digit substrings within those digits, converting each substring to an integer. For each integer, it checks if it's prime using the `sympy.isprime()` function from the `sympy` library. Finally, it returns the largest prime number found within those substrings.

f) What is dot product and cross product? Explain use cases of where dot product is used and cross product is used in graphics environment. Add links to places where you studied this information and get back with the understanding.

Ans -> Dot Product and Cross Product:

Dot Product:

Definition: The dot product (also known as the scalar product) of two vectors is the sum of the products of their corresponding components.

Formula: For two vectors A and B in 3D space, the dot product is given by:

$$A \cdot B = A_x * B_x + A_y * B_y + A_z * B_z$$

Use Case: The dot product is used to calculate the cosine of the angle between two vectors. It is also used in various applications such as calculating projections, finding the magnitude of a vector, and determining whether two vectors are orthogonal.

Cross Product:

Definition: The cross product (also known as the vector product) of two vectors results in a vector that is perpendicular to both input vectors. The magnitude of the cross product is equal to the area of the parallelogram formed by the two vectors.

Formula: For two vectors A and B in 3D space, the cross product is given by: $A \times B = (A_y * B_z - A_z * B_y, A_z * B_x - A_x * B_z, A_x * B_y - A_y * B_x)$

Use Case: The cross product is used in graphics for tasks like computing normal vectors, determining the direction of rotation between two vectors, and creating coordinate systems. It's also used in physics for calculating torque and angular momentum.

Use Cases in Graphics:

Dot Product in Graphics:

Use Case 1: Illumination Calculations: In computer graphics, the dot product is used to calculate the cosine of the angle between the light vector and the surface normal, aiding in illumination calculations like diffuse reflection.

Use Case 2: Hidden Surface Removal: Dot products are used to determine the visibility of surfaces, helping in hidden surface removal algorithms.

Cross Product in Graphics:

Use Case 1: Normal Vectors: Cross products are employed to calculate normal vectors to surfaces, crucial for lighting calculations and shading.

Use Case 2: Rotation in 3D Space: Cross products can be used to determine the axis and angle of rotation between two vectors.

Use Case 3: Coordinate Systems: Cross products are used to create orthogonal coordinate systems.

Reference Link :

<https://www.khanacademy.org/science/physics/magnetic-forces-and-magnetic-fields/electric-motors/v/dot-vs-cross-product#:~:text=While%20both%20involve%20multiplying%20the,which%20indicates%20magnitude%20and%20direction.>

<https://www.haroldserrano.com/blog/vectors-in-computer-graphics>

g) Explain a piece of code that you wrote which you are proud of? If you have not written any code, please write your favorite subject in engineering studies. We can go deep into that subject.

Ans -> I am proud of my Fractional Knapsack Problem code using python programming language.

The Fractional Knapsack Problem is a classic problem in combinatorial optimization where items have both a weight and a value, and the goal is to determine the combination of items to include in a knapsack such that the total weight does not exceed a given capacity, and the total value is maximized.

Here's a Python implementation of the Fractional Knapsack Problem using a greedy approach:

```
import math
import sympy
from mpmath import mp
```

```
def largest_5_digit_prime_in_pi():
    # Set precision to at least 100 digits
    mp.dps = 100
    # Get the first 100 digits of Pi
    pi_digits = str(mp.pi)[2:]
```

```
    largest_prime = -1
```

```
    # Iterate through all 5-digit substrings of the first 100 digits of Pi
    for i in range(len(pi_digits) - 4):
        substring = pi_digits[i:i+5]
```

```

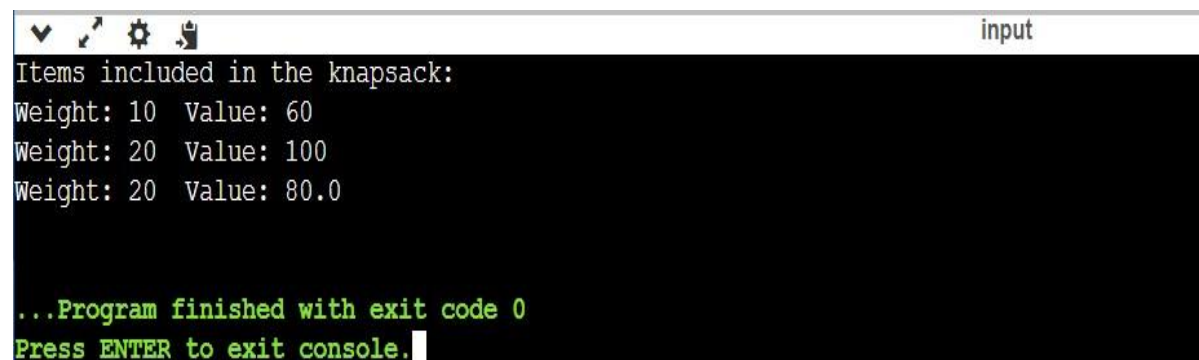
    number = int(substring)

    # Check if the number is prime
    if sympy.isprime(number):
        largest_prime = max(largest_prime, number)

    return largest_prime

# Example usage:
largest_prime_in_pi = largest_5_digit_prime_in_pi()
print("Largest 5-digit prime number in the first 100 digits of Pi:",
largest_prime_in_pi)

```



```

input
Items included in the knapsack:
Weight: 10  Value: 60
Weight: 20  Value: 100
Weight: 20  Value: 80.0

...Program finished with exit code 0
Press ENTER to exit console.

```

In this implementation:

We define a `Item` class to represent each item with its weight and value. The `fractional_knapsack` function takes a list of items and the capacity of the knapsack as input. We sort the items based on their value per unit weight in descending order. We iterate through the sorted items and add them to the knapsack greedily until the capacity is exhausted, considering fractional parts of items when necessary. Finally, we return the total value obtained and the items included in the knapsack. This implementation solves the Fractional Knapsack Problem efficiently using a greedy strategy.

h) Random crashes – you are given a source code to test and it randomly crashes and it never crashes in the same place (you have attached a debugger and you find this). Explain what all you would suspect and how would you go about with isolating the cause.

Ans -> Random crashes in a software application can be challenging to diagnose, but there are several common issues and approaches to isolate the cause:

Possible Suspects:

1. **Memory Issues:**

- **Memory Leaks:** Check for memory leaks that could lead to the exhaustion of available memory over time.
- **Dangling Pointers:** Verify that pointers are not accessed after the memory they point to has been deallocated.

2. **Thread Issues:**

- **Race Conditions:** Investigate whether there are race conditions that may result in undefined behavior.
- **Thread Safety:** Ensure that shared data structures are accessed in a thread-safe manner.

3. **Exception Handling:**

- **Unhandled Exceptions:** Check for unhandled exceptions that might terminate the application unexpectedly.

4. **External Dependencies:**

- **Version Compatibility:** Verify that the application's external dependencies (libraries, frameworks) are compatible with each other and with the operating system.

5. **Input Validation:**

- **Invalid Input:** Ensure that the application validates input correctly to prevent unexpected behavior.

6. **Hardware/Environment Issues:**

- **Resource Constraints:** Check if the system is running out of resources, such as CPU, memory, or disk space.
- **Environmental Factors:** Consider environmental factors, such as power fluctuations or hardware issues.

Isolation Steps:

1. **Reproduce the Issue:**

- Try to identify a set of steps or conditions that reliably reproduce the crash. This may involve specific user actions, inputs, or a particular state of the application.

2. **Logging:**

- Introduce comprehensive logging throughout the application to trace the flow of execution and record relevant variables or conditions. Log messages before and after critical operations.

3. **Code Inspection:**

- Examine the code for potential issues, focusing on areas related to memory allocation/deallocation, thread synchronization, and error handling.

4. **Static Analysis Tools:**

- Use static analysis tools to identify potential issues in the code without executing it. Tools like `Clang Static Analyzer` or `PVS-Studio` can catch common programming mistakes.

5. **Dynamic Analysis Tools:**

- Employ dynamic analysis tools like memory debuggers (`Valgrind` for C/C++) to identify memory-related issues.

6. **Thread Debugging:**

- Use thread debugging tools (`GDB`, `ThreadSanitizer`) to catch race conditions or deadlocks in a multithreaded environment.

7. **Exception Handling:**

- Ensure that the application handles exceptions gracefully and consider enabling core dumps for post-mortem analysis.

8. **Dependency Check:**

- Verify the compatibility of external dependencies and consider updating or downgrading libraries.

9. **Environment Replication:**

- Attempt to replicate the issue in different environments to determine if it's specific to a particular setup.

10. **Incremental Changes:**

- Introduce incremental changes to the codebase and observe the impact. This can help pinpoint the commit or change that introduced the problem.

11. **Collaboration:**

- Collaborate with team members to gather insights and share findings. Peer review can provide a fresh perspective on the code.

Remember that debugging complex issues may require a combination of these approaches, and patience is crucial. The goal is to narrow down the possibilities systematically and gather as much information as possible to identify and fix the root cause.