Lab Report

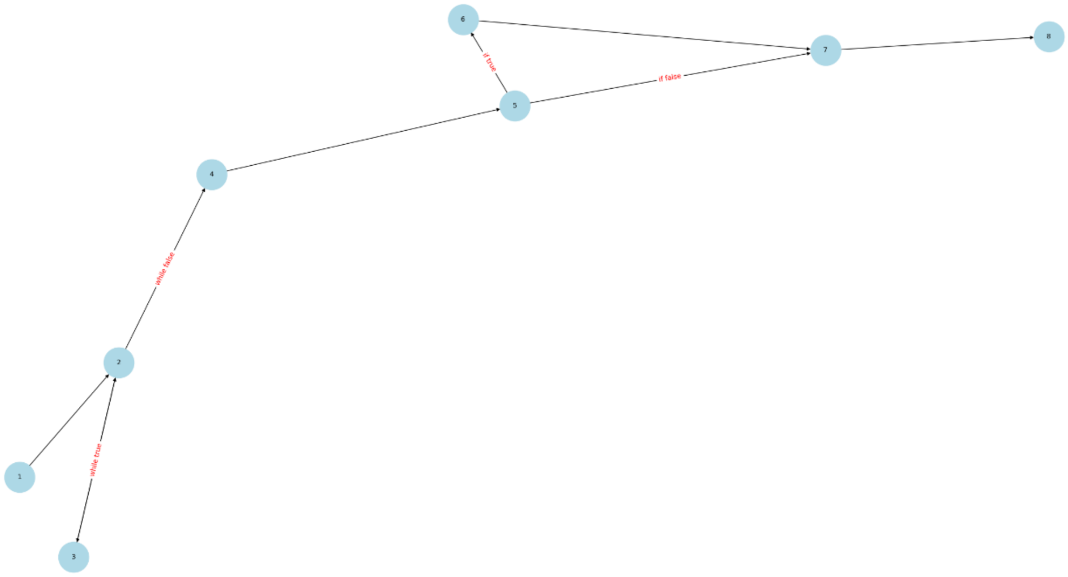
By: Jahmil Ally (501045419)

Observations and Analysis

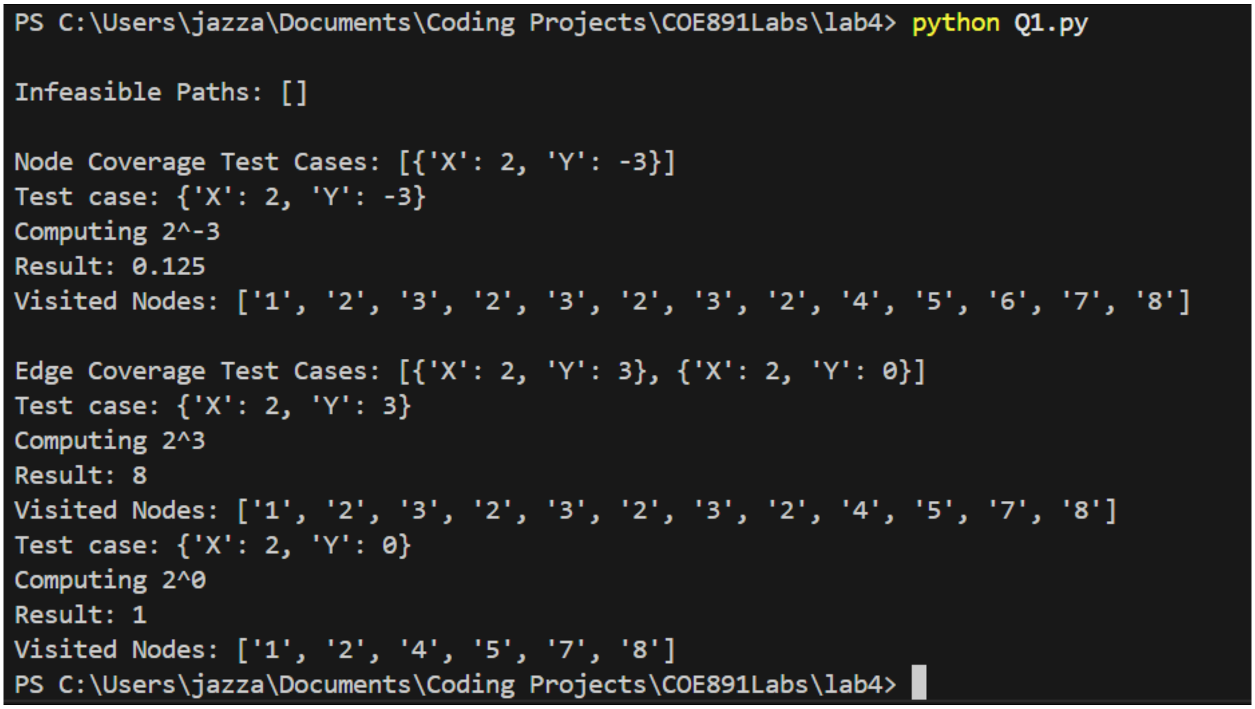
Q1:

Data Flow Graph:

Node	Lines
1	1,2,3,4
2	5
3	6, 7
4	8
5	9
6	10
7	11
8	12, 13



Output:

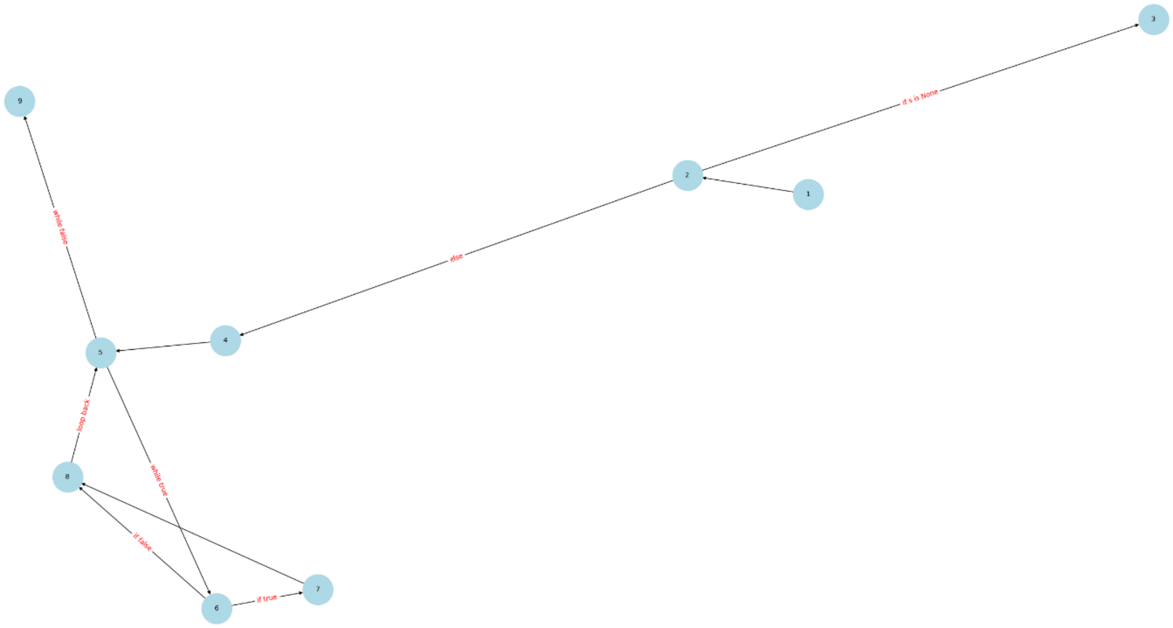


```
PS C:\Users\jazza\Documents\Coding Projects\COE891Labs\lab4> python Q1.py

Infeasible Paths: []

Node Coverage Test Cases: [{'X': 2, 'Y': -3}]
Test case: {'X': 2, 'Y': -3}
Computing 2^-3
Result: 0.125
Visited Nodes: ['1', '2', '3', '2', '3', '2', '3', '2', '4', '5', '6', '7', '8']

Edge Coverage Test Cases: [{'X': 2, 'Y': 3}, {'X': 2, 'Y': 0}]
Test case: {'X': 2, 'Y': 3}
Computing 2^3
Result: 8
Visited Nodes: ['1', '2', '3', '2', '3', '2', '3', '2', '4', '5', '7', '8']
Test case: {'X': 2, 'Y': 0}
Computing 2^0
Result: 1
Visited Nodes: ['1', '2', '4', '5', '7', '8']
PS C:\Users\jazza\Documents\Coding Projects\COE891Labs\lab4>
```

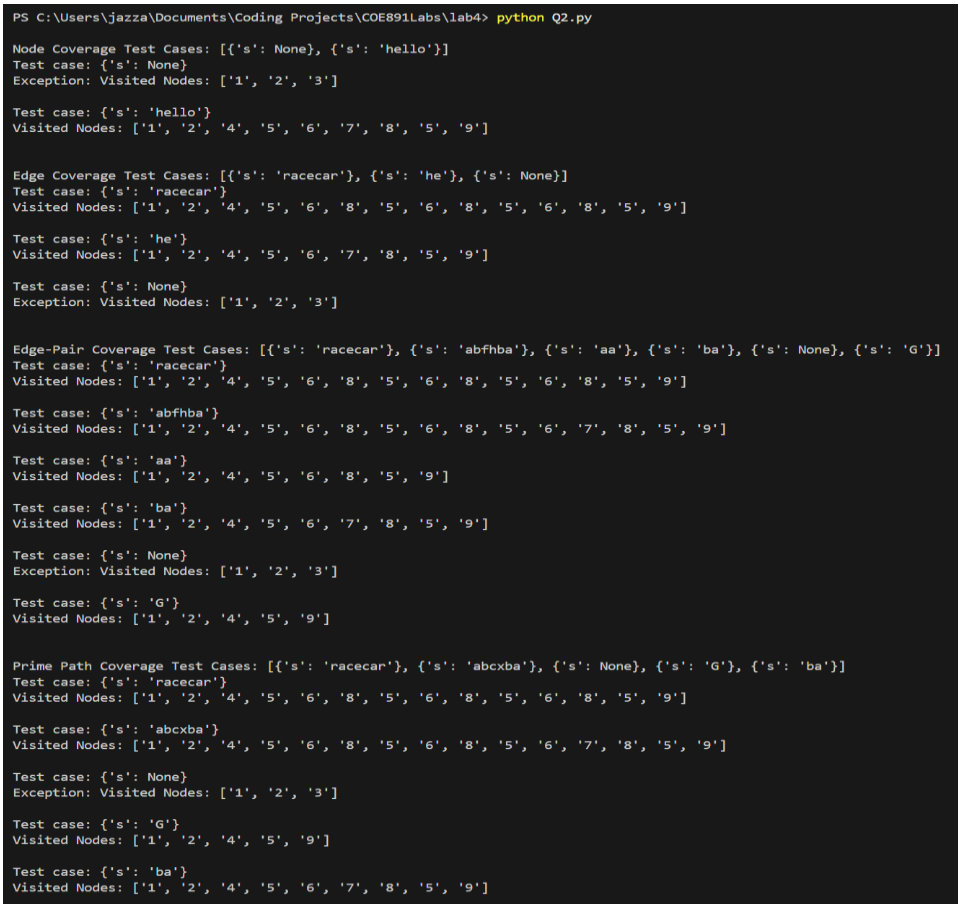
Q2:

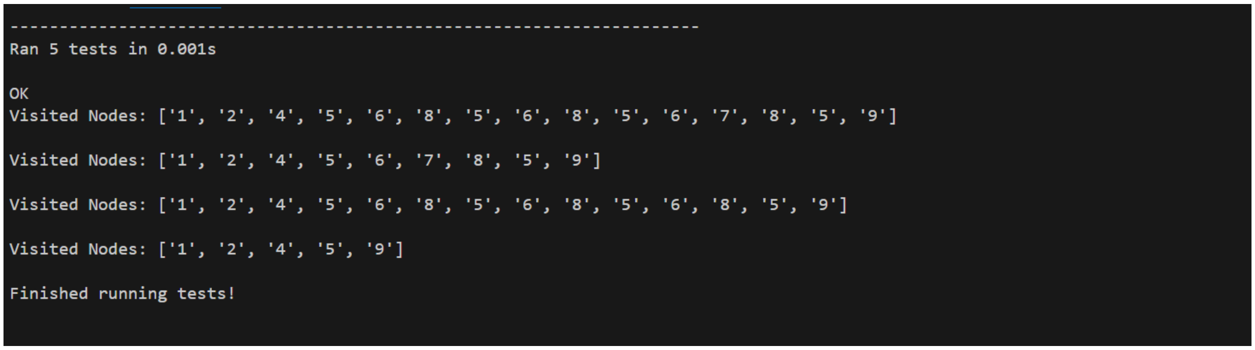
Data Flow Graph:

Node	Lines
1	1
2	2
3	3
4	4, 5, 6
5	7
6	8
7	9
8	10, 11
9	12, 13



Output:


```
PS C:\Users\jazza\Documents\Coding Projects\COE891Labs\lab4> python Q2.py

Node Coverage Test Cases: [{'s': None}, {'s': 'hello'}]
Test case: {'s': None}
Exception: Visited Nodes: ['1', '2', '3']

Test case: {'s': 'hello'}
Visited Nodes: ['1', '2', '4', '5', '6', '7', '8', '5', '9']


Edge Coverage Test Cases: [{'s': 'racecar'}, {'s': 'he'}, {'s': None}]
Test case: {'s': 'racecar'}
Visited Nodes: ['1', '2', '4', '5', '6', '8', '5', '6', '8', '5', '6', '8', '5', '9']

Test case: {'s': 'he'}
Visited Nodes: ['1', '2', '4', '5', '6', '7', '8', '5', '9']

Test case: {'s': None}
Exception: Visited Nodes: ['1', '2', '3']


Edge-Pair Coverage Test Cases: [{'s': 'racecar'}, {'s': 'abfhba'}, {'s': 'aa'}, {'s': 'ba'}, {'s': None}, {'s': 'G'}]
Test case: {'s': 'racecar'}
Visited Nodes: ['1', '2', '4', '5', '6', '8', '5', '6', '8', '5', '6', '8', '5', '9']

Test case: {'s': 'abfhba'}
Visited Nodes: ['1', '2', '4', '5', '6', '8', '5', '6', '8', '5', '6', '7', '8', '5', '9']

Test case: {'s': 'aa'}
Visited Nodes: ['1', '2', '4', '5', '6', '8', '5', '9']

Test case: {'s': 'ba'}
Visited Nodes: ['1', '2', '4', '5', '6', '7', '8', '5', '9']

Test case: {'s': None}
Exception: Visited Nodes: ['1', '2', '3']

Test case: {'s': 'G'}
Visited Nodes: ['1', '2', '4', '5', '9']


Prime Path Coverage Test Cases: [{'s': 'racecar'}, {'s': 'abcxba'}, {'s': None}, {'s': 'G'}, {'s': 'ba'}]
Test case: {'s': 'racecar'}
Visited Nodes: ['1', '2', '4', '5', '6', '8', '5', '6', '8', '5', '6', '8', '5', '9']

Test case: {'s': 'abcxba'}
Visited Nodes: ['1', '2', '4', '5', '6', '8', '5', '6', '8', '5', '6', '7', '8', '5', '9']

Test case: {'s': None}
Exception: Visited Nodes: ['1', '2', '3']

Test case: {'s': 'G'}
Visited Nodes: ['1', '2', '4', '5', '9']

Test case: {'s': 'ba'}
Visited Nodes: ['1', '2', '4', '5', '6', '7', '8', '5', '9']
```

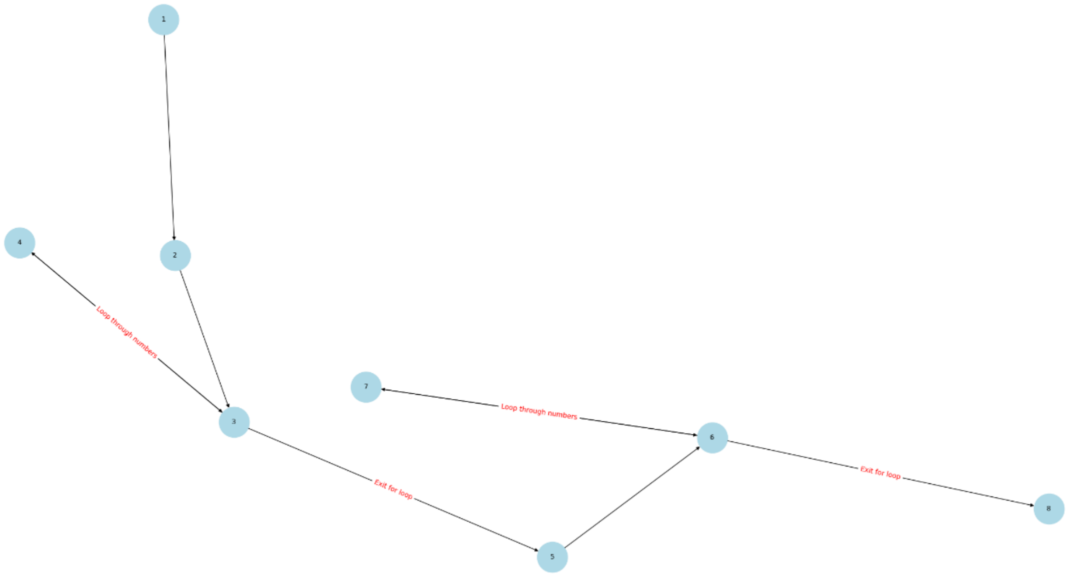


```
----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
Visited Nodes: ['1', '2', '4', '5', '6', '8', '5', '6', '8', '5', '6', '7', '8', '5', '9']

Visited Nodes: ['1', '2', '4', '5', '6', '7', '8', '5', '9']

Visited Nodes: ['1', '2', '4', '5', '6', '8', '5', '6', '8', '5', '6', '8', '5', '9']

Visited Nodes: ['1', '2', '4', '5', '9']

Finished running tests!
```

Q3:

1. Control flow and data flow coverage graph

Node	Lines
1	1
2	2, 3,4
3	5
4	6
5	7, 8, 9
6	10
7	11
8	12 to 19



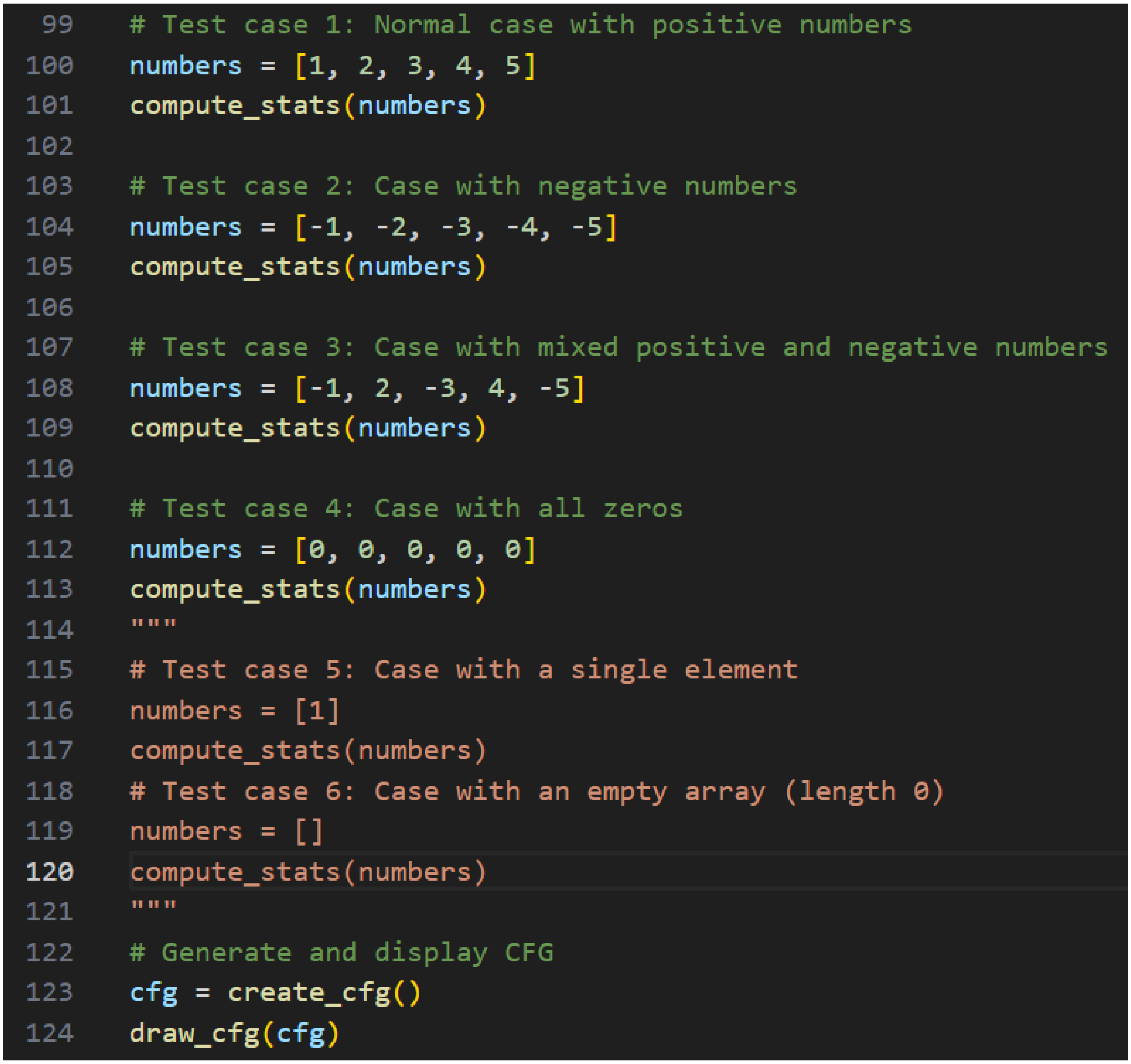
2. DU Pairs (Node Pairs) for Each Variable
   DU pairs are pairs of nodes where a variable is defined (D) and used (U).
   - length:
     - Defined at node 2

    - Used at nodes 3, 5, 6, 8
- sum_values:
    - Defined at node 2
    - Used at nodes 4, 5
- numbers_sorted:
    - Defined at node 5
    - Used at node 5
- median:
    - Defined at node 5
    - Used at node 8
- mean:
    - Defined at node 5
    - Used at nodes 7, 8
- varsum:
    - Defined at node 5
    - Used at nodes 7, 8
- variance:
    - Defined at node 8
    - Used at node 8
- standard_deviation:
    - Defined at node 8
    - Used at node 8

3. DU Paths for Each DU Pair for Each Variable
   DU paths are paths from a definition to a use of a variable.
    - length:
        - (2, 3): length is defined at node 2 and used at node 3
        - (2, 5): length is defined at node 2 and used at node 5
        - (2, 6): length is defined at node 2 and used at node 6
        - (2, 8): length is defined at node 2 and used at node 8
    - sum_values:
        - (2, 4): sum_values is defined at node 2 and used at node 4
        - (2, 5): sum_values is defined at node 2 and used at node 5
    - numbers_sorted:
        - (5, 5): numbers_sorted is defined and used at node 5
    - median:
        - (5, 8): median is defined at node 5 and used at node 8
    - mean:
        - (5, 7): mean is defined at node 5 and used at node 7
        - (5, 8): mean is defined at node 5 and used at node 8

- varsum:
    - (5, 7): varsum is defined at node 5 and used at node 7
    - (5, 8): varsum is defined at node 5 and used at node 8
- variance:
    - (8, 8): variance is defined and used at node 8
- standard_deviation:
    - (8, 8): standard_deviation is defined and used at node 8

4. Test cases to cover du paths



```python
# Test case 1: Normal case with positive numbers
numbers = [1, 2, 3, 4, 5]
compute_stats(numbers)

# Test case 2: Case with negative numbers
numbers = [-1, -2, -3, -4, -5]
compute_stats(numbers)

# Test case 3: Case with mixed positive and negative numbers
numbers = [-1, 2, -3, 4, -5]
compute_stats(numbers)

# Test case 4: Case with all zeros
numbers = [0, 0, 0, 0, 0]
compute_stats(numbers)
"""
# Test case 5: Case with a single element
numbers = [1]
compute_stats(numbers)
# Test case 6: Case with an empty array (length 0)
numbers = []
compute_stats(numbers)
"""
# Generate and display CFG
cfg = create_cfg()
draw_cfg(cfg)
```

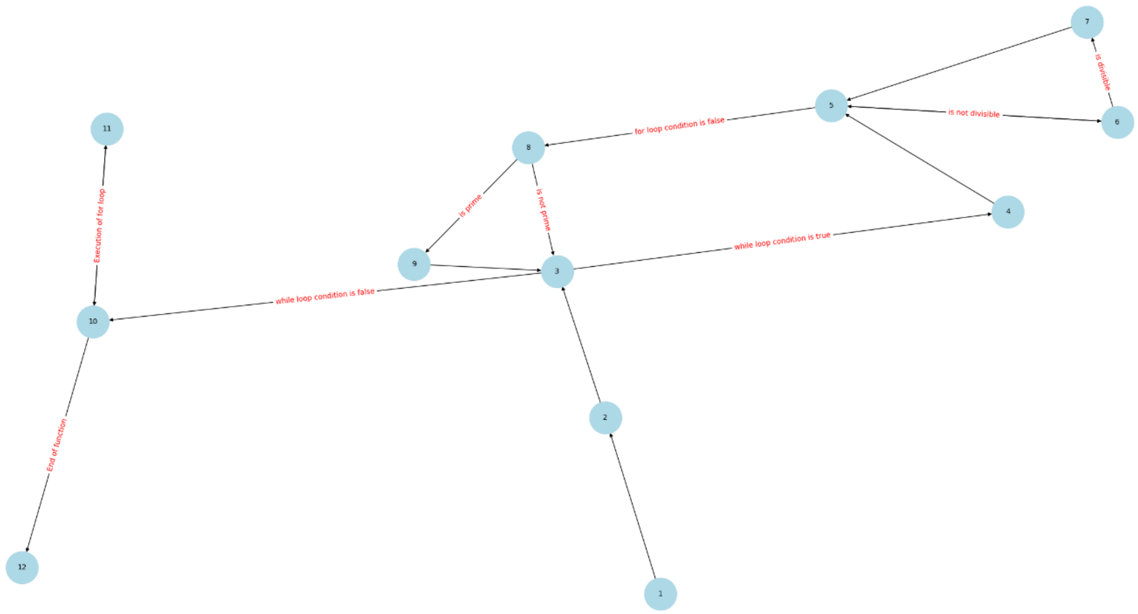
5. Can’t run all test cases to cover du paths

- IndexError: This occurs because the program tries to access an element in an empty list. Specifically, numbers_sorted[length // 2] tries to access an element at index 0, which does not exist in an empty list.
- ZeroDivisionError: This occurs because the program tries to divide by zero. Specifically, mean = sum_values / length and variance = varsum / (length - 1) both involve division by zero when length is 0 or 1.

Q4:

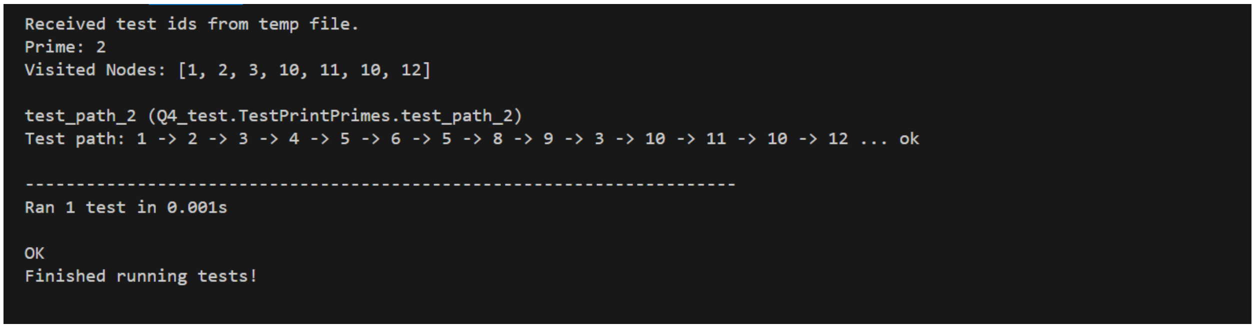
1. Control flow and data flow coverage graph

Node	Lines
1	1, 2, 3, 4
2	5 to 12
3	13
4	14, 15
5	16
6	17
7	18 to 22
8	23
9	24, to 27
10	28
11	29
12	30



2. Test Case: n = 1
   When n = 1, the initial value of num_primes is 1. The while loop condition num_primes < n will be 1 < 1, which is false. Therefore, the while loop body will not be executed, and the control will directly move to the for loop that prints the primes

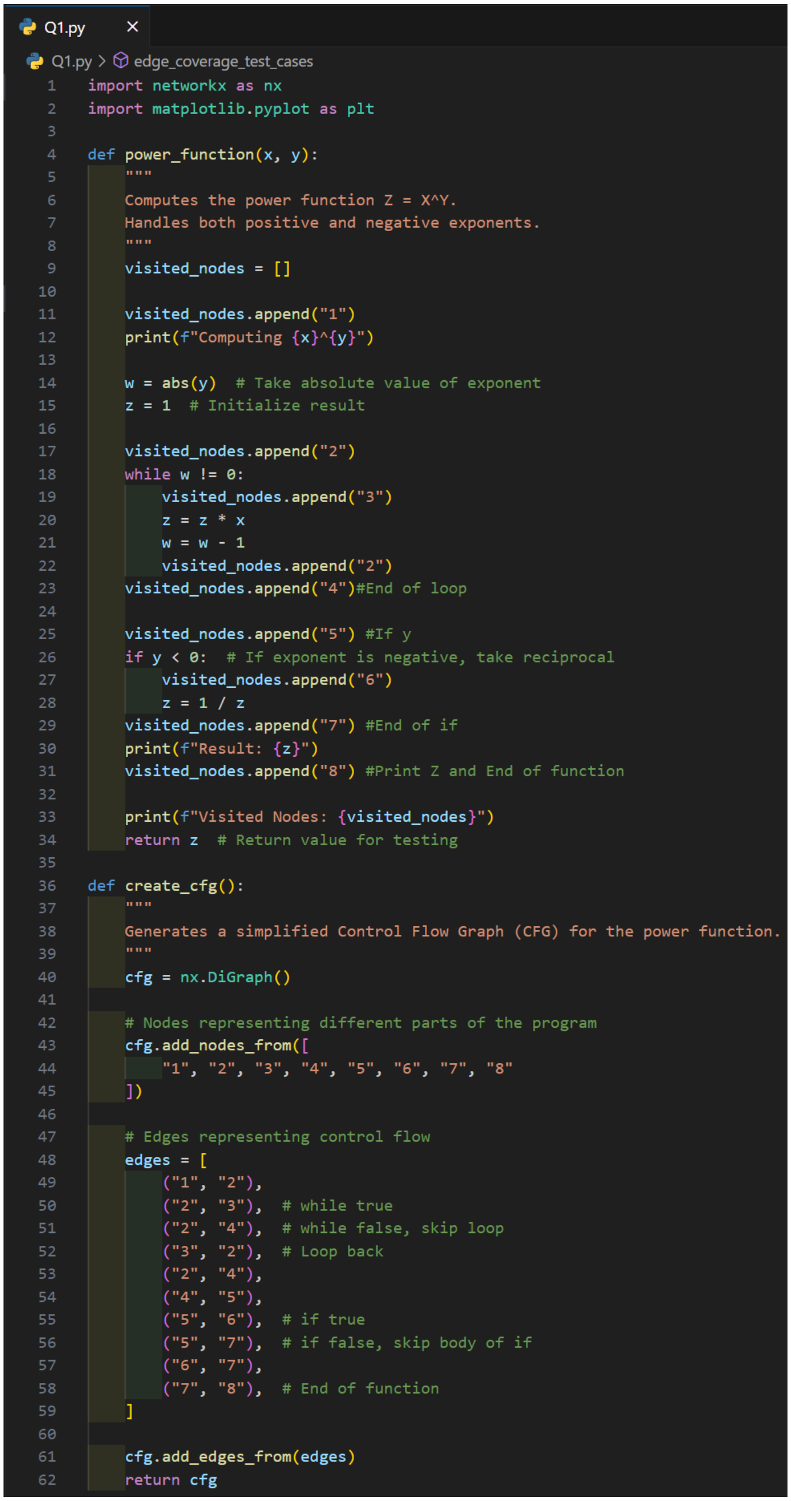
3. Test Paths for Edge Coverage but Not Prime Path Coverage
   - Input: n = 1
     - Path 1: 1 -> 2 -> 3 -> 10 -> 11 -> 10 -> 12
     - This path covers the edges from the start to the initialization, checks the while loop condition (which is false), and then moves to the for loop to print the primes.
   - Input: n = 2
     - Path 2: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 5 -> 8 -> 9 -> 3 -> 10 -> 11 -> 10 -> 12
     - This path covers the edges from the start to the initialization, checks the while loop condition (which is true), executes the while loop body, checks the for loop condition, finds a prime, and then moves to the for loop to print the primes.
4. Unit Tests



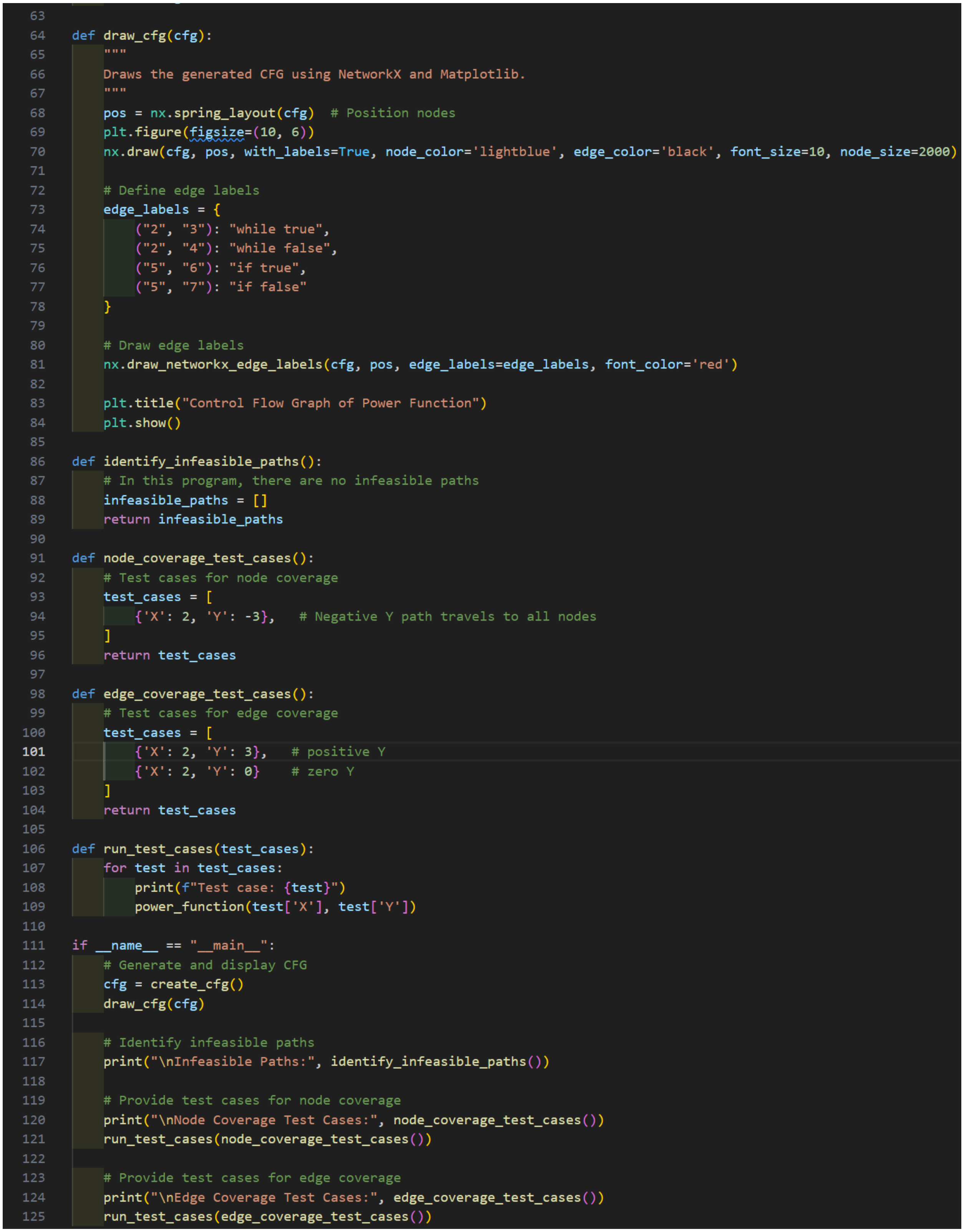
```
Received test ids from temp file.
Prime: 2
Visited Nodes: [1, 2, 3, 10, 11, 10, 12]

test_path_2 (Q4_test.TestPrintPrimes.test_path_2)
Test path: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 5 -> 8 -> 9 -> 3 -> 10 -> 11 -> 10 -> 12 ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Finished running tests!
```

Appendix:

Q1:

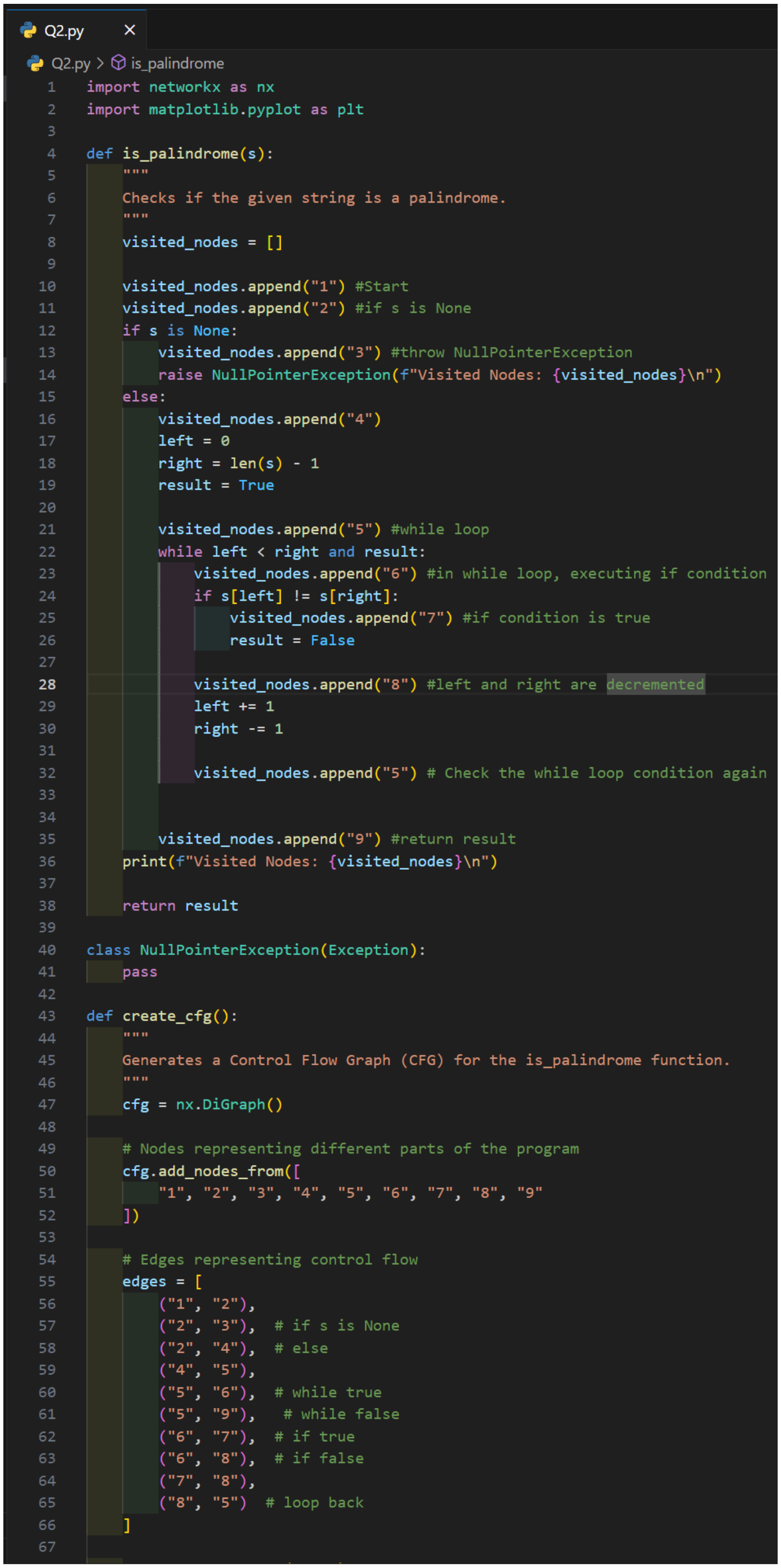


```python
import networkx as nx
import matplotlib.pyplot as plt

def power_function(x, y):
    """
    Computes the power function Z = X^Y.
    Handles both positive and negative exponents.
    """
    visited_nodes = []

    visited_nodes.append("1")
    print(f"Computing {x}^{y}")

    w = abs(y)  # Take absolute value of exponent
    z = 1  # Initialize result

    visited_nodes.append("2")
    while w != 0:
        visited_nodes.append("3")
        z = z * x
        w = w - 1
        visited_nodes.append("2")
    visited_nodes.append("4")#End of loop

    visited_nodes.append("5") #If y
    if y < 0:  # If exponent is negative, take reciprocal
        visited_nodes.append("6")
        z = 1 / z
    visited_nodes.append("7") #End of if
    print(f"Result: {z}")
    visited_nodes.append("8") #Print Z and End of function

    print(f"Visited Nodes: {visited_nodes}")
    return z  # Return value for testing

def create_cfg():
    """
    Generates a simplified Control Flow Graph (CFG) for the power function.
    """
    cfg = nx.DiGraph()

    # Nodes representing different parts of the program
    cfg.add_nodes_from([
        "1", "2", "3", "4", "5", "6", "7", "8"
    ])

    # Edges representing control flow
    edges = [
        ("1", "2"),
        ("2", "3"),  # while true
        ("2", "4"),  # while false, skip loop
        ("3", "2"),  # Loop back
        ("2", "4"),
        ("4", "5"),
        ("5", "6"),  # if true
        ("5", "7"),  # if false, skip body of if
        ("6", "7"),
        ("7", "8"),  # End of function
    ]

    cfg.add_edges_from(edges)
    return cfg
```

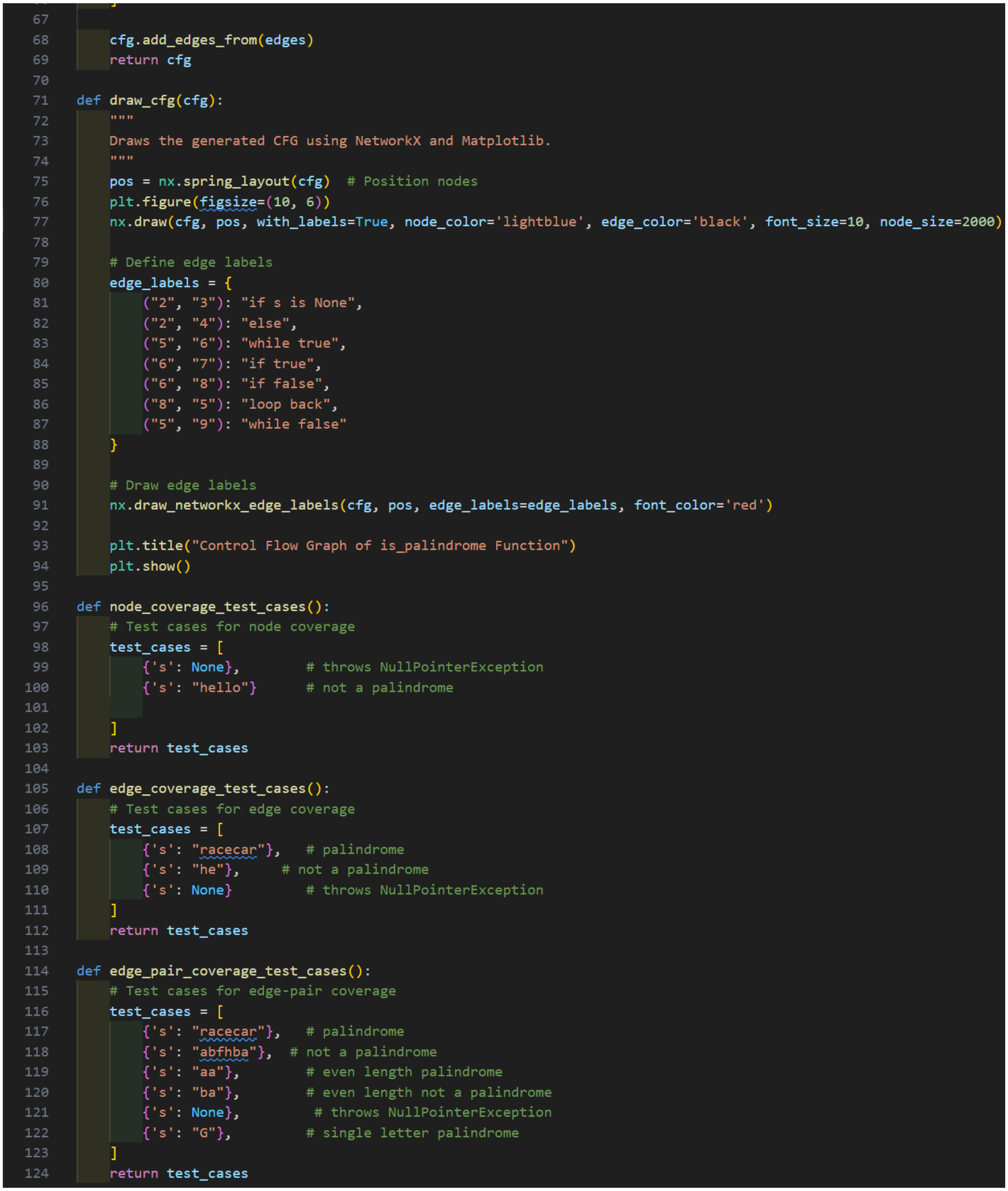


```python

def draw_cfg(cfg):
    """
    Draws the generated CFG using NetworkX and Matplotlib.
    """
    pos = nx.spring_layout(cfg)  # Position nodes
    plt.figure(figsize=(10, 6))
    nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)

    # Define edge labels
    edge_labels = {
        ("2", "3"): "while true",
        ("2", "4"): "while false",
        ("5", "6"): "if true",
        ("5", "7"): "if false"
    }

    # Draw edge labels
    nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')

    plt.title("Control Flow Graph of Power Function")
    plt.show()

def identify_infeasible_paths():
    # In this program, there are no infeasible paths
    infeasible_paths = []
    return infeasible_paths

def node_coverage_test_cases():
    # Test cases for node coverage
    test_cases = [
        {'X': 2, 'Y': -3},   # Negative Y path travels to all nodes
    ]
    return test_cases

def edge_coverage_test_cases():
    # Test cases for edge coverage
    test_cases = [
        {'X': 2, 'Y': 3},    # positive Y
        {'X': 2, 'Y': 0}     # zero Y
    ]
    return test_cases

def run_test_cases(test_cases):
    for test in test_cases:
        print(f"Test case: {test}")
        power_function(test['X'], test['Y'])

if __name__ == "__main__":
    # Generate and display CFG
    cfg = create_cfg()
    draw_cfg(cfg)

    # Identify infeasible paths
    print("\nInfeasible Paths:", identify_infeasible_paths())

    # Provide test cases for node coverage
    print("\nNode Coverage Test Cases:", node_coverage_test_cases())
    run_test_cases(node_coverage_test_cases())

    # Provide test cases for edge coverage
    print("\nEdge Coverage Test Cases:", edge_coverage_test_cases())
    run_test_cases(edge_coverage_test_cases())
```

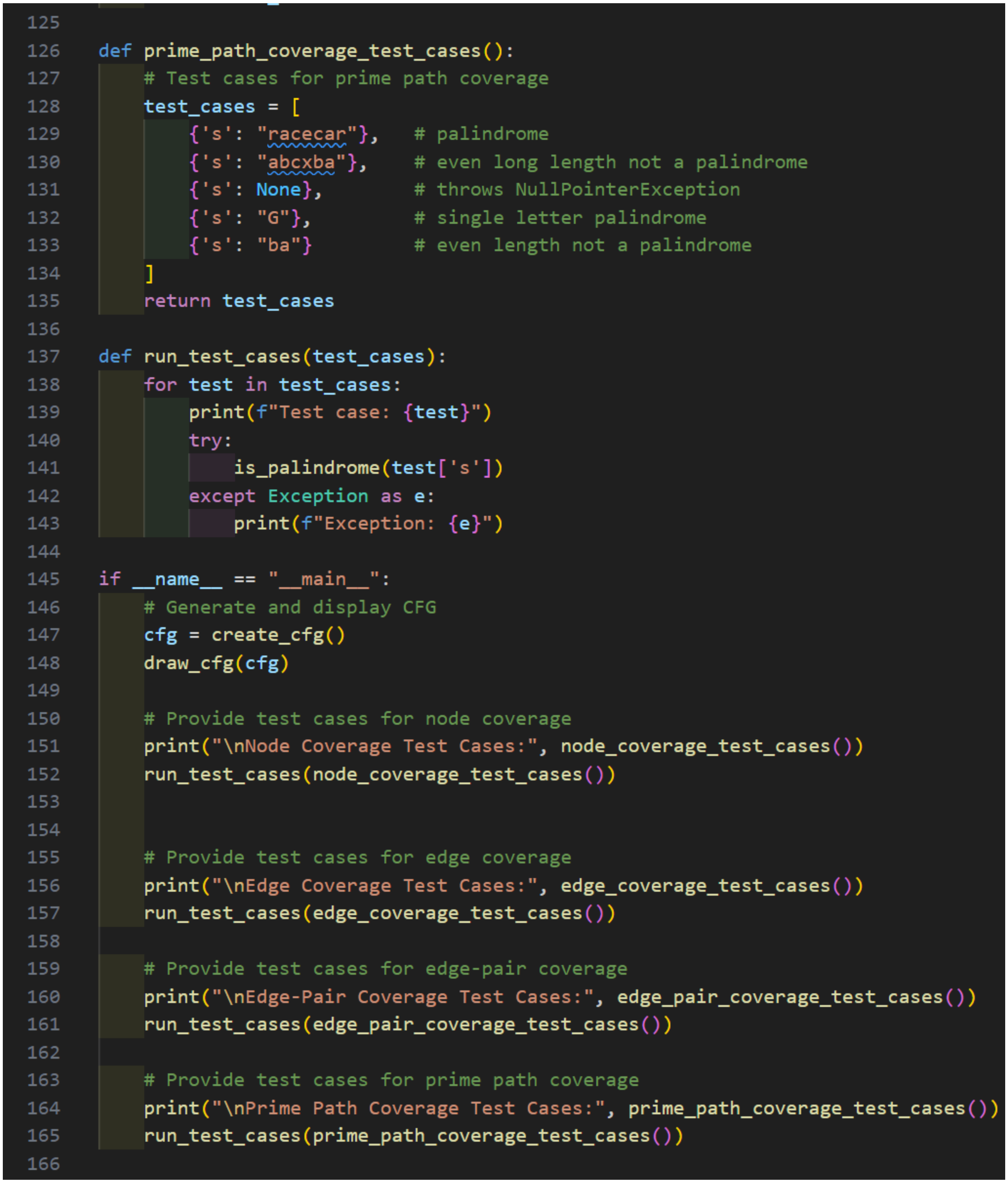
Q2:



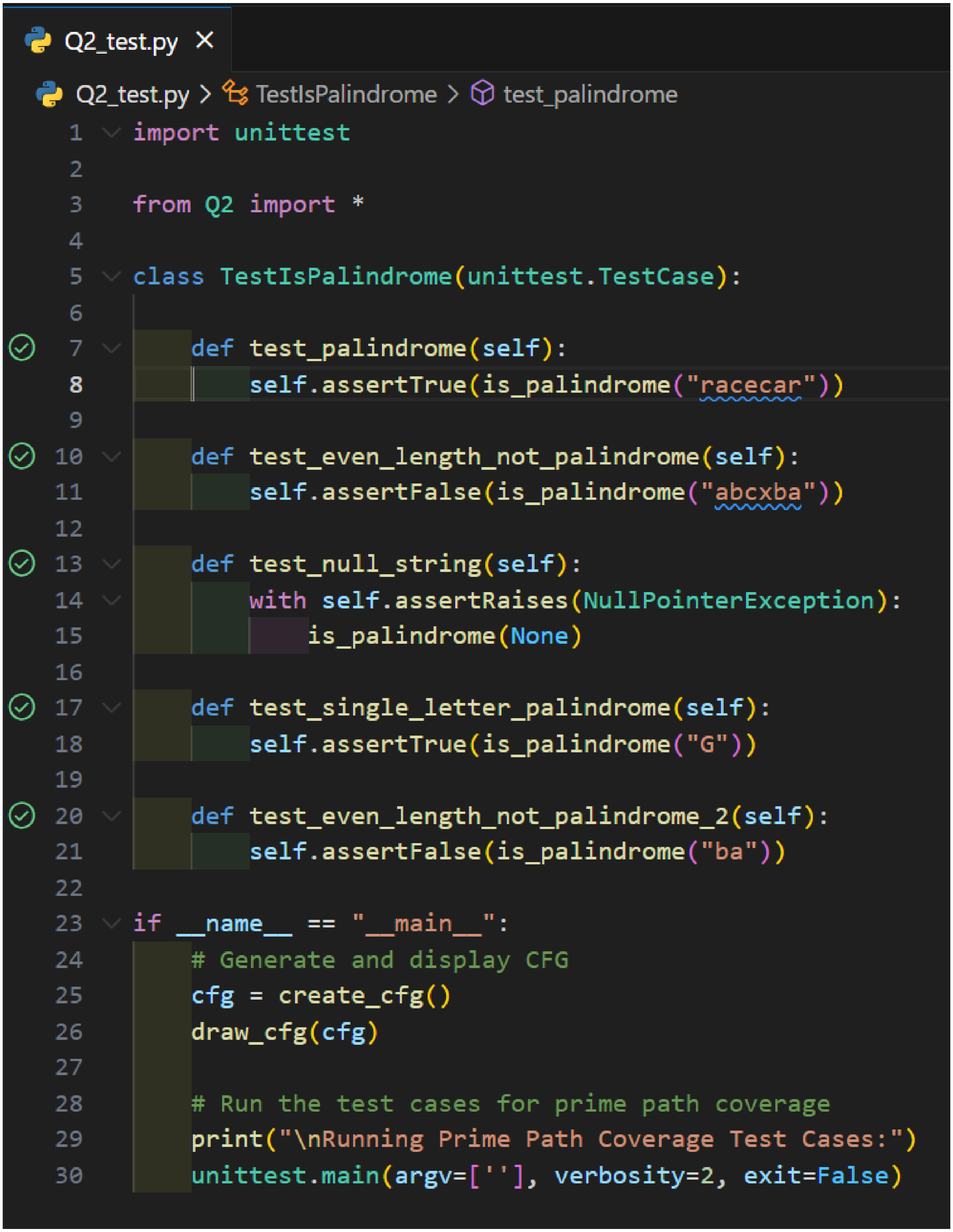
```python
import networkx as nx
import matplotlib.pyplot as plt

def is_palindrome(s):
    """
    Checks if the given string is a palindrome.
    """
    visited_nodes = []

    visited_nodes.append("1") #Start
    visited_nodes.append("2") #if s is None
    if s is None:
        visited_nodes.append("3") #throw NullPointerException
        raise NullPointerException(f"Visited Nodes: {visited_nodes}\n")
    else:
        visited_nodes.append("4")
        left = 0
        right = len(s) - 1
        result = True

        visited_nodes.append("5") #while loop
        while left < right and result:
            visited_nodes.append("6") #in while loop, executing if condition
            if s[left] != s[right]:
                visited_nodes.append("7") #if condition is true
                result = False

            visited_nodes.append("8") #left and right are decremented
            left += 1
            right -= 1

            visited_nodes.append("5") # Check the while loop condition again


        visited_nodes.append("9") #return result
    print(f"Visited Nodes: {visited_nodes}\n")

    return result

class NullPointerException(Exception):
    pass

def create_cfg():
    """
    Generates a Control Flow Graph (CFG) for the is_palindrome function.
    """
    cfg = nx.DiGraph()

    # Nodes representing different parts of the program
    cfg.add_nodes_from([
        "1", "2", "3", "4", "5", "6", "7", "8", "9"
    ])

    # Edges representing control flow
    edges = [
        ("1", "2"),
        ("2", "3"),  # if s is None
        ("2", "4"),  # else
        ("4", "5"),
        ("5", "6"),  # while true
        ("5", "9"),   # while false
        ("6", "7"),  # if true
        ("6", "8"),  # if false
        ("7", "8"),
        ("8", "5")  # loop back
    ]

```



```python
    ]

    cfg.add_edges_from(edges)
    return cfg

def draw_cfg(cfg):
    """
    Draws the generated CFG using NetworkX and Matplotlib.
    """
    pos = nx.spring_layout(cfg)  # Position nodes
    plt.figure(figsize=(10, 6))
    nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)

    # Define edge labels
    edge_labels = {
        ("2", "3"): "if s is None",
        ("2", "4"): "else",
        ("5", "6"): "while true",
        ("6", "7"): "if true",
        ("6", "8"): "if false",
        ("8", "5"): "loop back",
        ("5", "9"): "while false"
    }

    # Draw edge labels
    nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')

    plt.title("Control Flow Graph of is_palindrome Function")
    plt.show()

def node_coverage_test_cases():
    # Test cases for node coverage
    test_cases = [
        {'s': None},        # throws NullPointerException
        {'s': "hello"}      # not a palindrome

    ]
    return test_cases

def edge_coverage_test_cases():
    # Test cases for edge coverage
    test_cases = [
        {'s': "racecar"},   # palindrome
        {'s': "he"},     # not a palindrome
        {'s': None}         # throws NullPointerException
    ]
    return test_cases

def edge_pair_coverage_test_cases():
    # Test cases for edge-pair coverage
    test_cases = [
        {'s': "racecar"},   # palindrome
        {'s': "abfhba"},  # not a palindrome
        {'s': "aa"},        # even length palindrome
        {'s': "ba"},        # even length not a palindrome
        {'s': None},         # throws NullPointerException
        {'s': "G"},         # single letter palindrome
    ]
    return test_cases
```

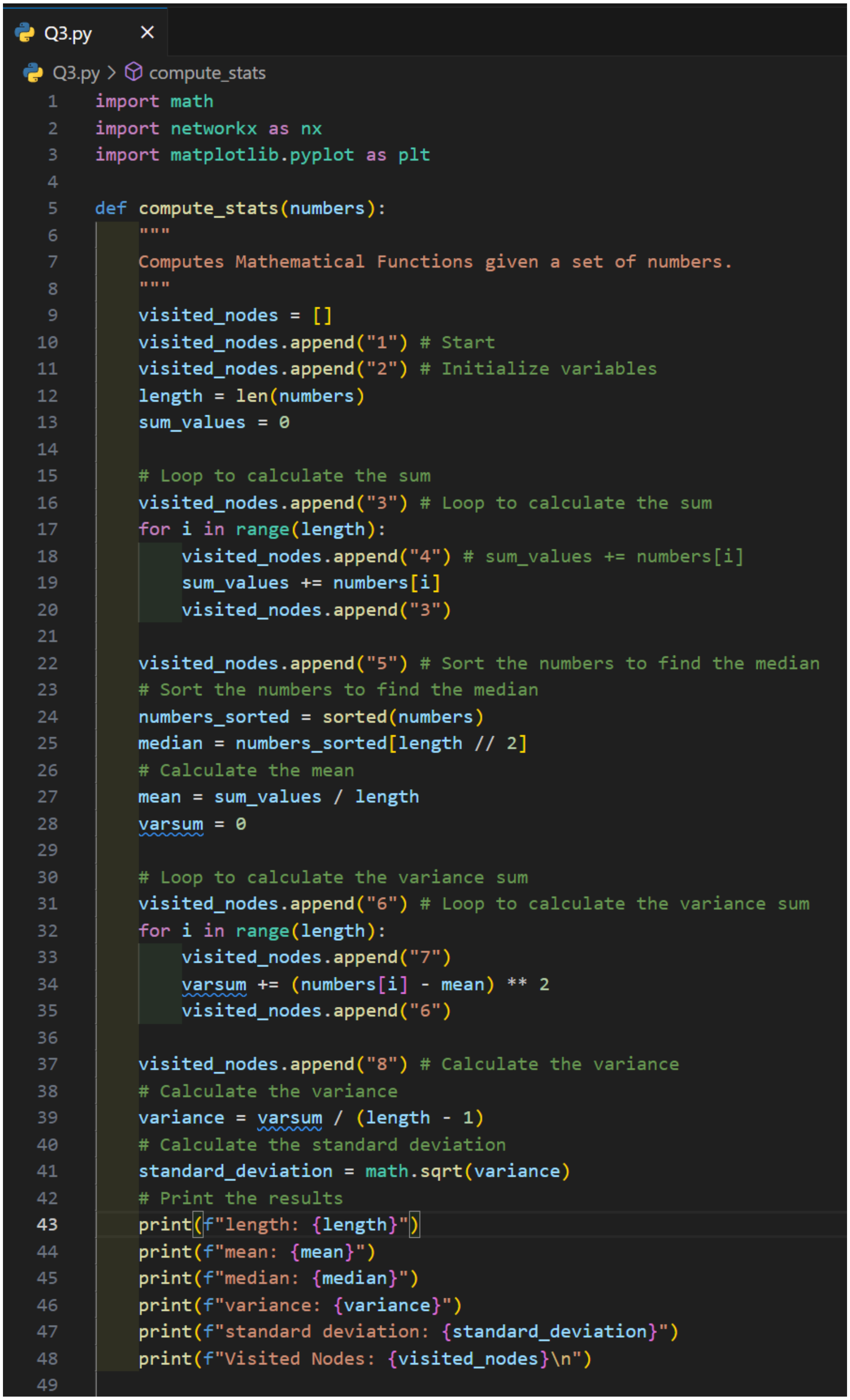


```python

def prime_path_coverage_test_cases():
    # Test cases for prime path coverage
    test_cases = [
        {'s': "racecar"},   # palindrome
        {'s': "abcxba"},    # even long length not a palindrome
        {'s': None},        # throws NullPointerException
        {'s': "G"},         # single letter palindrome
        {'s': "ba"}         # even length not a palindrome
    ]
    return test_cases

def run_test_cases(test_cases):
    for test in test_cases:
        print(f"Test case: {test}")
        try:
            is_palindrome(test['s'])
        except Exception as e:
            print(f"Exception: {e}")

if __name__ == "__main__":
    # Generate and display CFG
    cfg = create_cfg()
    draw_cfg(cfg)

    # Provide test cases for node coverage
    print("\nNode Coverage Test Cases:", node_coverage_test_cases())
    run_test_cases(node_coverage_test_cases())


    # Provide test cases for edge coverage
    print("\nEdge Coverage Test Cases:", edge_coverage_test_cases())
    run_test_cases(edge_coverage_test_cases())

    # Provide test cases for edge-pair coverage
    print("\nEdge-Pair Coverage Test Cases:", edge_pair_coverage_test_cases())
    run_test_cases(edge_pair_coverage_test_cases())

    # Provide test cases for prime path coverage
    print("\nPrime Path Coverage Test Cases:", prime_path_coverage_test_cases())
    run_test_cases(prime_path_coverage_test_cases())

```

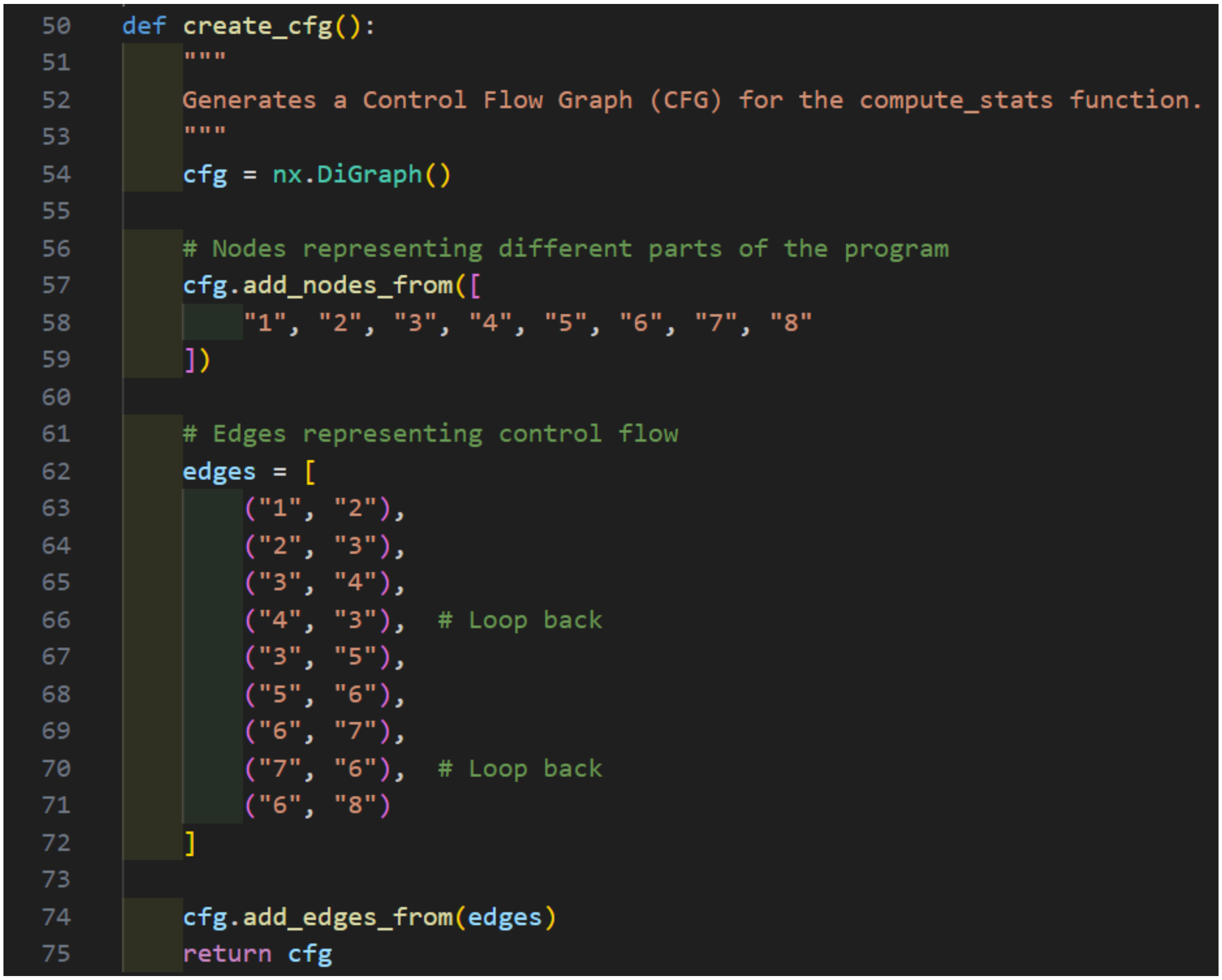


```python
import unittest

from Q2 import *

class TestIsPalindrome(unittest.TestCase):

    def test_palindrome(self):
        self.assertTrue(is_palindrome("racecar"))

    def test_even_length_not_palindrome(self):
        self.assertFalse(is_palindrome("abcxba"))

    def test_null_string(self):
        with self.assertRaises(NullPointerException):
            is_palindrome(None)

    def test_single_letter_palindrome(self):
        self.assertTrue(is_palindrome("G"))

    def test_even_length_not_palindrome_2(self):
        self.assertFalse(is_palindrome("ba"))

if __name__ == "__main__":
    # Generate and display CFG
    cfg = create_cfg()
    draw_cfg(cfg)

    # Run the test cases for prime path coverage
    print("\nRunning Prime Path Coverage Test Cases:")
    unittest.main(argv=[''], verbosity=2, exit=False)
```

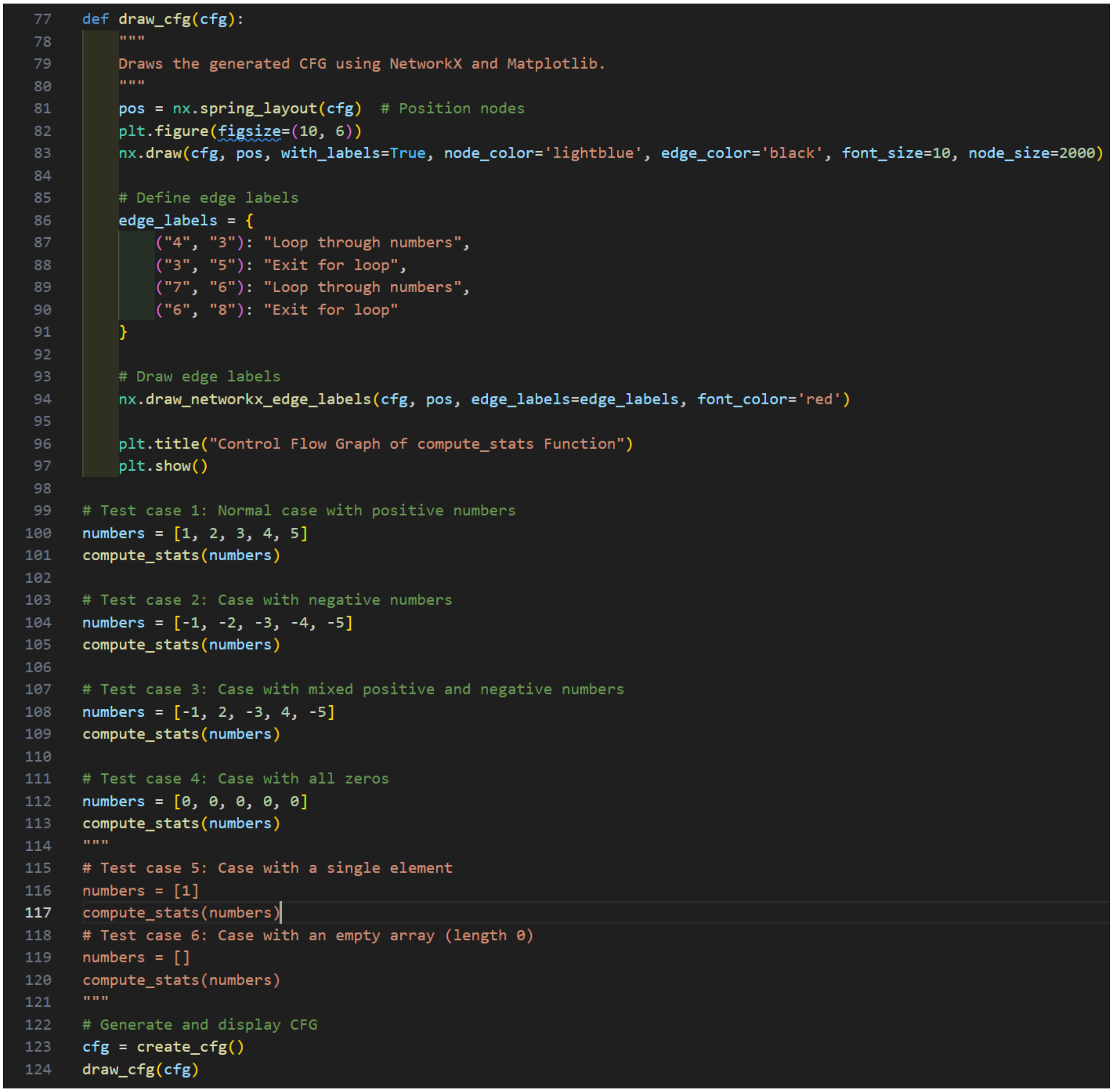
Q3:



```python
import math
import networkx as nx
import matplotlib.pyplot as plt

def compute_stats(numbers):
    """
    Computes Mathematical Functions given a set of numbers.
    """
    visited_nodes = []
    visited_nodes.append("1") # Start
    visited_nodes.append("2") # Initialize variables
    length = len(numbers)
    sum_values = 0

    # Loop to calculate the sum
    visited_nodes.append("3") # Loop to calculate the sum
    for i in range(length):
        visited_nodes.append("4") # sum_values += numbers[i]
        sum_values += numbers[i]
        visited_nodes.append("3")

    visited_nodes.append("5") # Sort the numbers to find the median
    # Sort the numbers to find the median
    numbers_sorted = sorted(numbers)
    median = numbers_sorted[length // 2]
    # Calculate the mean
    mean = sum_values / length
    varsum = 0

    # Loop to calculate the variance sum
    visited_nodes.append("6") # Loop to calculate the variance sum
    for i in range(length):
        visited_nodes.append("7")
        varsum += (numbers[i] - mean) ** 2
        visited_nodes.append("6")

    visited_nodes.append("8") # Calculate the variance
    # Calculate the variance
    variance = varsum / (length - 1)
    # Calculate the standard deviation
    standard_deviation = math.sqrt(variance)
    # Print the results
    print(f"length: {length}")
    print(f"mean: {mean}")
    print(f"median: {median}")
    print(f"variance: {variance}")
    print(f"standard deviation: {standard_deviation}")
    print(f"Visited Nodes: {visited_nodes}\n")

```

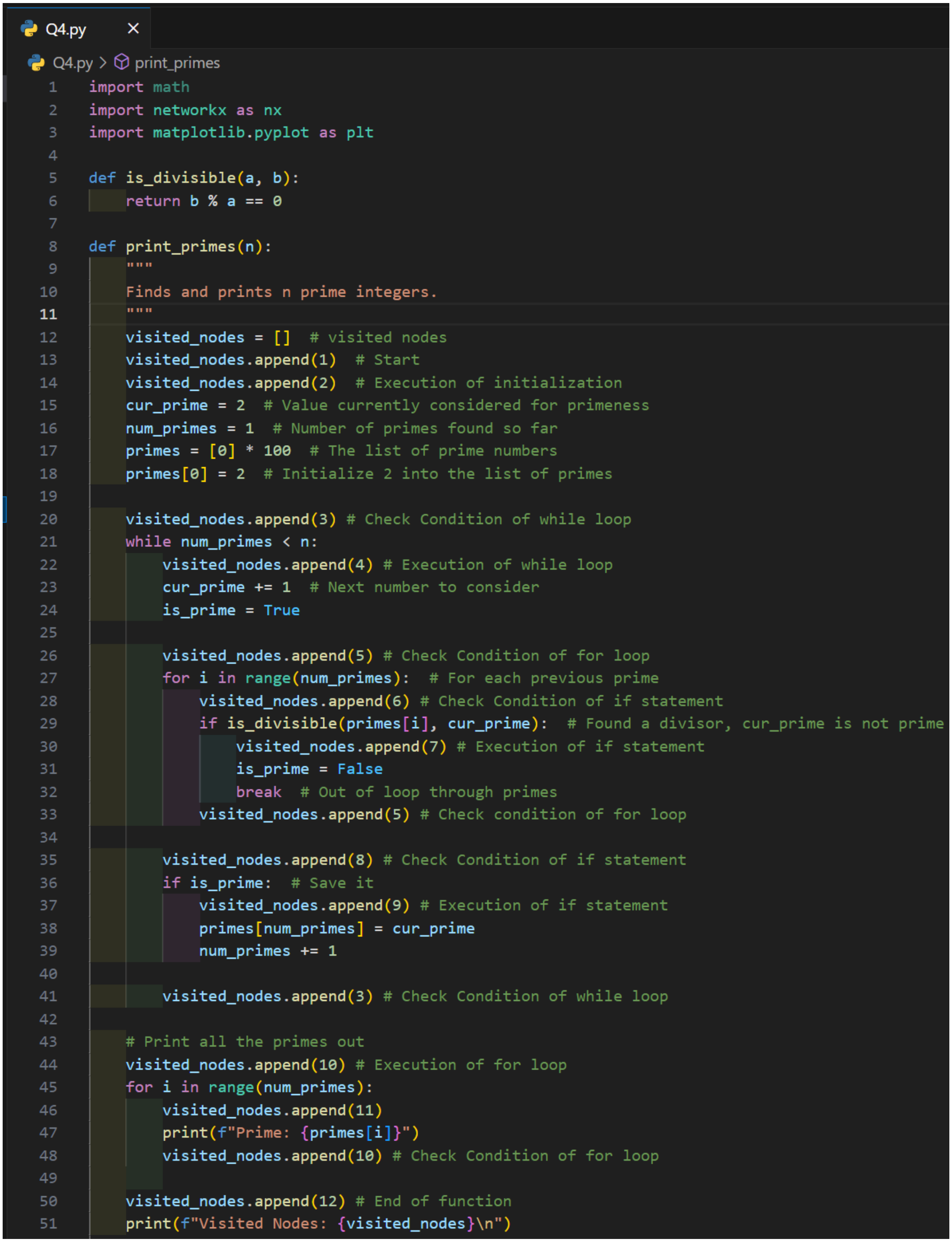


```python
def create_cfg():
    """
    Generates a Control Flow Graph (CFG) for the compute_stats function.
    """
    cfg = nx.DiGraph()

    # Nodes representing different parts of the program
    cfg.add_nodes_from([
        "1", "2", "3", "4", "5", "6", "7", "8"
    ])

    # Edges representing control flow
    edges = [
        ("1", "2"),
        ("2", "3"),
        ("3", "4"),
        ("4", "3"),  # Loop back
        ("3", "5"),
        ("5", "6"),
        ("6", "7"),
        ("7", "6"),  # Loop back
        ("6", "8")
    ]

    cfg.add_edges_from(edges)
    return cfg
```

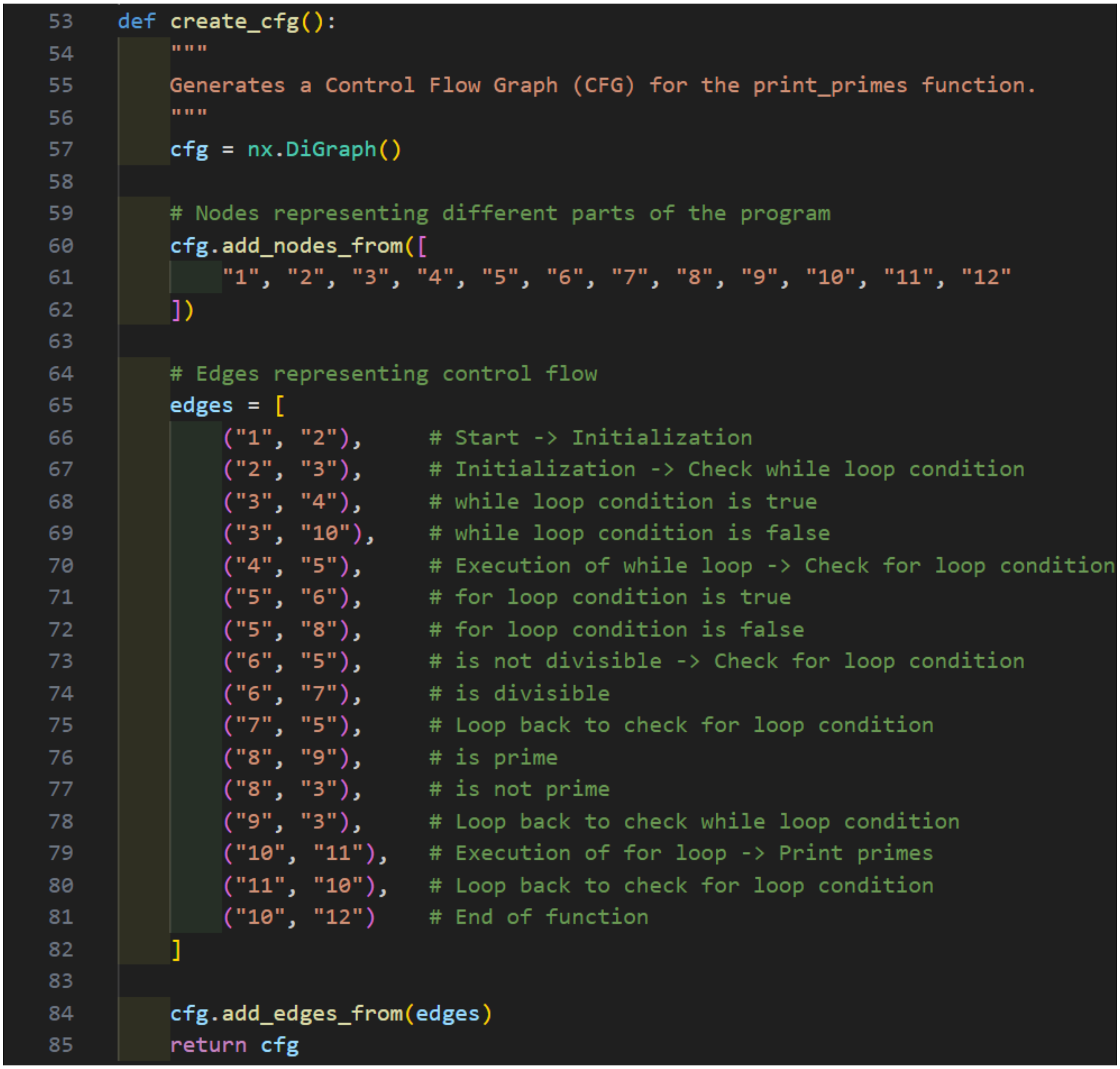


```python
def draw_cfg(cfg):
    """
    Draws the generated CFG using NetworkX and Matplotlib.
    """
    pos = nx.spring_layout(cfg)  # Position nodes
    plt.figure(figsize=(10, 6))
    nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)

    # Define edge labels
    edge_labels = {
        ("4", "3"): "Loop through numbers",
        ("3", "5"): "Exit for loop",
        ("7", "6"): "Loop through numbers",
        ("6", "8"): "Exit for loop"
    }

    # Draw edge labels
    nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')

    plt.title("Control Flow Graph of compute_stats Function")
    plt.show()

# Test case 1: Normal case with positive numbers
numbers = [1, 2, 3, 4, 5]
compute_stats(numbers)

# Test case 2: Case with negative numbers
numbers = [-1, -2, -3, -4, -5]
compute_stats(numbers)

# Test case 3: Case with mixed positive and negative numbers
numbers = [-1, 2, -3, 4, -5]
compute_stats(numbers)

# Test case 4: Case with all zeros
numbers = [0, 0, 0, 0, 0]
compute_stats(numbers)
"""
# Test case 5: Case with a single element
numbers = [1]
compute_stats(numbers)
# Test case 6: Case with an empty array (length 0)
numbers = []
compute_stats(numbers)
"""
# Generate and display CFG
cfg = create_cfg()
draw_cfg(cfg)
```

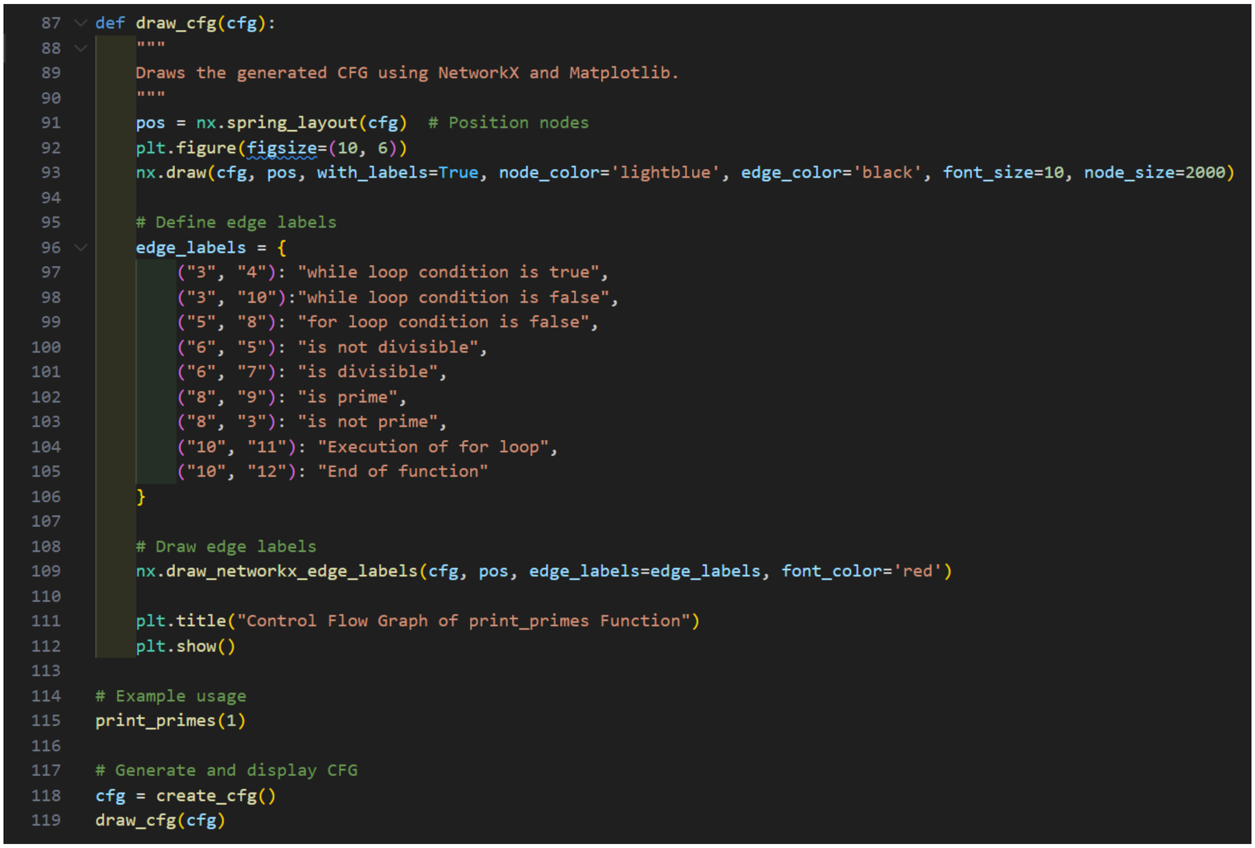
Q4:



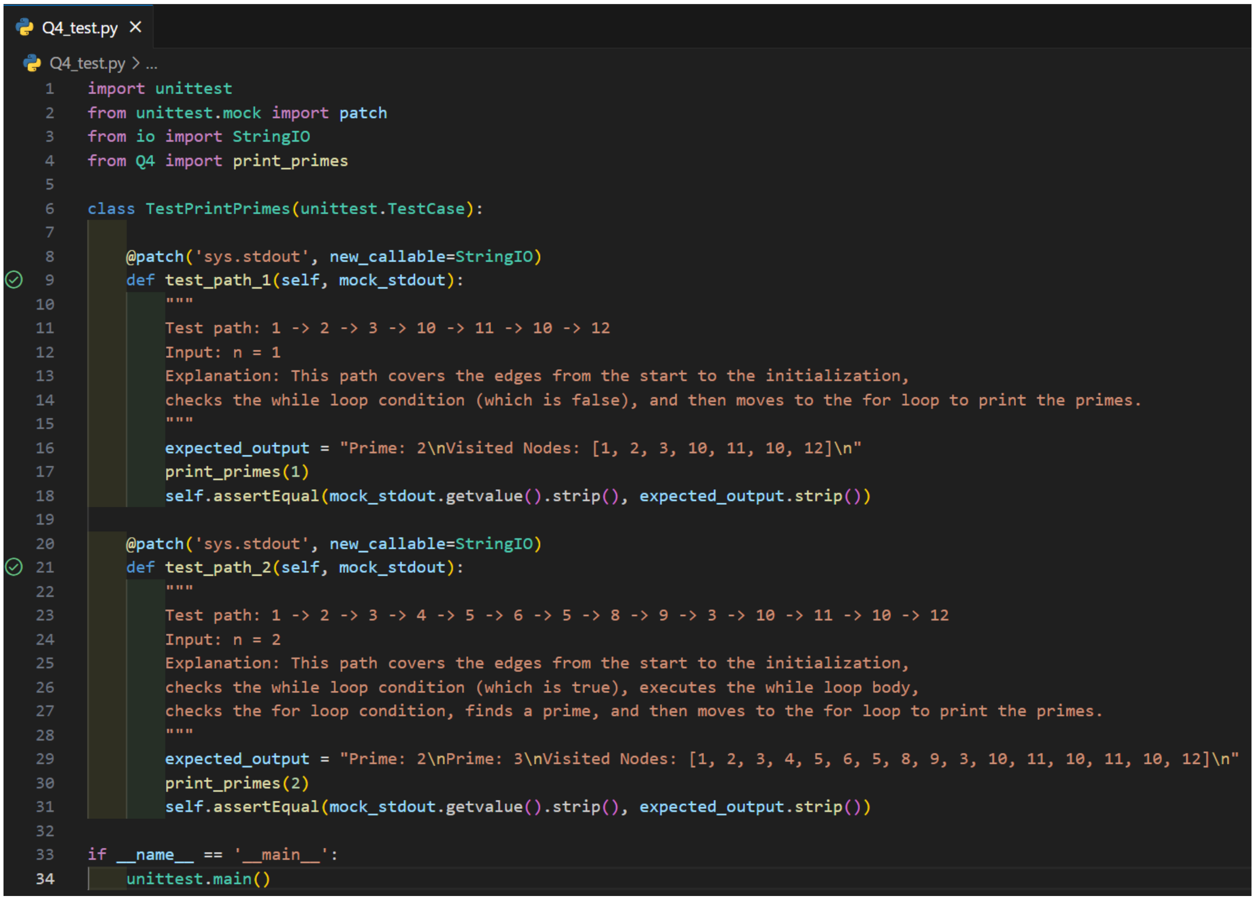
```python
import math
import networkx as nx
import matplotlib.pyplot as plt

def is_divisible(a, b):
    return b % a == 0

def print_primes(n):
    """
    Finds and prints n prime integers.
    """
    visited_nodes = []  # visited nodes
    visited_nodes.append(1)  # Start
    visited_nodes.append(2)  # Execution of initialization
    cur_prime = 2  # Value currently considered for primeness
    num_primes = 1  # Number of primes found so far
    primes = [0] * 100  # The list of prime numbers
    primes[0] = 2  # Initialize 2 into the list of primes

    visited_nodes.append(3) # Check Condition of while loop
    while num_primes < n:
        visited_nodes.append(4) # Execution of while loop
        cur_prime += 1  # Next number to consider
        is_prime = True

        visited_nodes.append(5) # Check Condition of for loop
        for i in range(num_primes):  # For each previous prime
            visited_nodes.append(6) # Check Condition of if statement
            if is_divisible(primes[i], cur_prime):  # Found a divisor, cur_prime is not prime
                visited_nodes.append(7) # Execution of if statement
                is_prime = False
                break  # Out of loop through primes
            visited_nodes.append(5) # Check condition of for loop

        visited_nodes.append(8) # Check Condition of if statement
        if is_prime:  # Save it
            visited_nodes.append(9) # Execution of if statement
            primes[num_primes] = cur_prime
            num_primes += 1

        visited_nodes.append(3) # Check Condition of while loop

    # Print all the primes out
    visited_nodes.append(10) # Execution of for loop
    for i in range(num_primes):
        visited_nodes.append(11)
        print(f"Prime: {primes[i]}")
        visited_nodes.append(10) # Check Condition of for loop

    visited_nodes.append(12) # End of function
    print(f"Visited Nodes: {visited_nodes}\n")
```



```python
def create_cfg():
    """
    Generates a Control Flow Graph (CFG) for the print_primes function.
    """
    cfg = nx.DiGraph()

    # Nodes representing different parts of the program
    cfg.add_nodes_from([
        "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12"
    ])

    # Edges representing control flow
    edges = [
        ("1", "2"),     # Start -> Initialization
        ("2", "3"),     # Initialization -> Check while loop condition
        ("3", "4"),     # while loop condition is true
        ("3", "10"),    # while loop condition is false
        ("4", "5"),     # Execution of while loop -> Check for loop condition
        ("5", "6"),     # for loop condition is true
        ("5", "8"),     # for loop condition is false
        ("6", "5"),     # is not divisible -> Check for loop condition
        ("6", "7"),     # is divisible
        ("7", "5"),     # Loop back to check for loop condition
        ("8", "9"),     # is prime
        ("8", "3"),     # is not prime
        ("9", "3"),     # Loop back to check while loop condition
        ("10", "11"),   # Execution of for loop -> Print primes
        ("11", "10"),   # Loop back to check for loop condition
        ("10", "12")    # End of function
    ]

    cfg.add_edges_from(edges)
    return cfg
```



```python
def draw_cfg(cfg):
    """
    Draws the generated CFG using NetworkX and Matplotlib.
    """
    pos = nx.spring_layout(cfg)  # Position nodes
    plt.figure(figsize=(10, 6))
    nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)

    # Define edge labels
    edge_labels = {
        ("3", "4"): "while loop condition is true",
        ("3", "10"):"while loop condition is false",
        ("5", "8"): "for loop condition is false",
        ("6", "5"): "is not divisible",
        ("6", "7"): "is divisible",
        ("8", "9"): "is prime",
        ("8", "3"): "is not prime",
        ("10", "11"): "Execution of for loop",
        ("10", "12"): "End of function"
    }

    # Draw edge labels
    nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')

    plt.title("Control Flow Graph of print_primes Function")
    plt.show()

# Example usage
print_primes(1)

# Generate and display CFG
cfg = create_cfg()
draw_cfg(cfg)
```



```python
import unittest
from unittest.mock import patch
from io import StringIO
from Q4 import print_primes

class TestPrintPrimes(unittest.TestCase):

    @patch('sys.stdout', new_callable=StringIO)
    def test_path_1(self, mock_stdout):
        """
        Test path: 1 -> 2 -> 3 -> 10 -> 11 -> 10 -> 12
        Input: n = 1
        Explanation: This path covers the edges from the start to the initialization,
        checks the while loop condition (which is false), and then moves to the for loop to print the primes.
        """
        expected_output = "Prime: 2\nVisited Nodes: [1, 2, 3, 10, 11, 10, 12]\n"
        print_primes(1)
        self.assertEqual(mock_stdout.getvalue().strip(), expected_output.strip())

    @patch('sys.stdout', new_callable=StringIO)
    def test_path_2(self, mock_stdout):
        """
        Test path: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 5 -> 8 -> 9 -> 3 -> 10 -> 11 -> 10 -> 12
        Input: n = 2
        Explanation: This path covers the edges from the start to the initialization,
        checks the while loop condition (which is true), executes the while loop body,
        checks the for loop condition, finds a prime, and then moves to the for loop to print the primes.
        """
        expected_output = "Prime: 2\nPrime: 3\nVisited Nodes: [1, 2, 3, 4, 5, 6, 5, 8, 9, 3, 10, 11, 10, 11, 10, 12]\n"
        print_primes(2)
        self.assertEqual(mock_stdout.getvalue().strip(), expected_output.strip())

if __name__ == '__main__':
    unittest.main()
```