

Lab Report

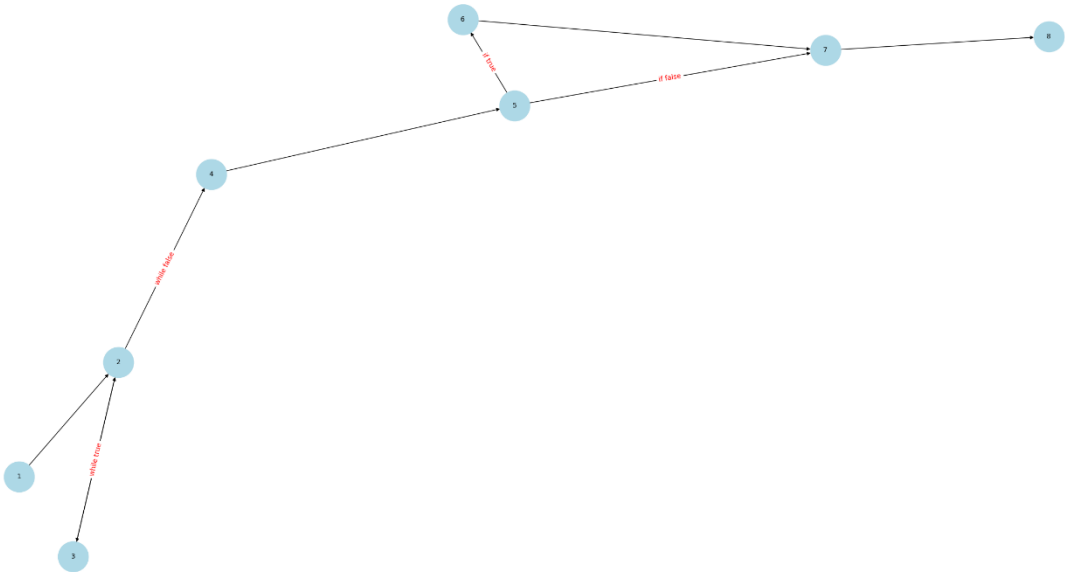
By: Jahmil Ally (501045419)

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']

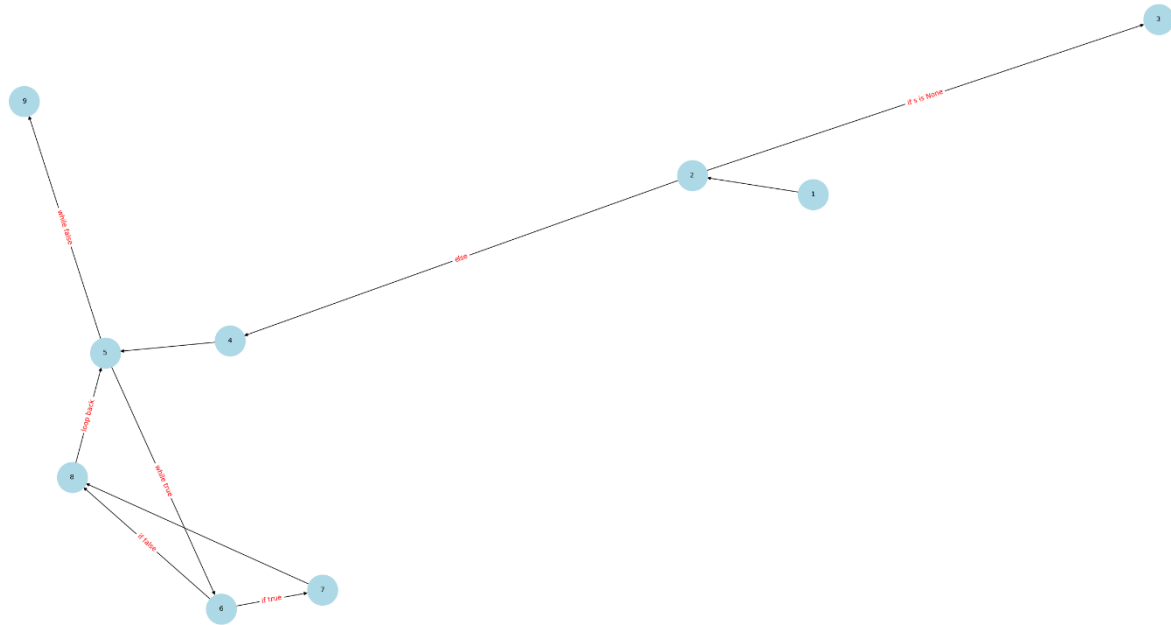
Edge Coverage Test Cases: [{'X': 2, 'Y': 3}, {'X': 2, 'Y': 0}]
Test case: {'X': 2, 'Y': 3}
Computing 2^3
Result: 8
Visited Nodes: ['1', '2', '3', '2', '3', '2', '3', '2', '4', '5', '7', '8']
Test case: {'X': 2, 'Y': 0}
Computing 2^0
Result: 1
Visited Nodes: ['1', '2', '4', '5', '7', '8']
PS C:\Users\jazza\Documents\Coding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

```

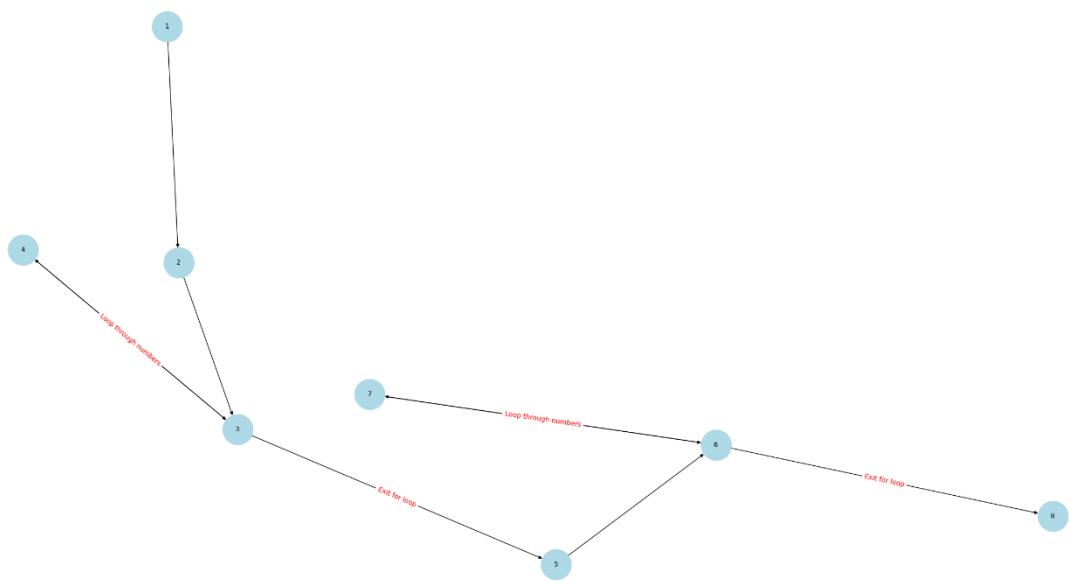
```
-----
Ran 5 tests in 0.001s

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

```

99  # Test case 1: Normal case with positive numbers
100 numbers = [1, 2, 3, 4, 5]
101 compute_stats(numbers)
102
103 # Test case 2: Case with negative numbers
104 numbers = [-1, -2, -3, -4, -5]
105 compute_stats(numbers)
106
107 # Test case 3: Case with mixed positive and negative numbers
108 numbers = [-1, 2, -3, 4, -5]
109 compute_stats(numbers)
110
111 # Test case 4: Case with all zeros
112 numbers = [0, 0, 0, 0, 0]
113 compute_stats(numbers)
114 """
115 # Test case 5: Case with a single element
116 numbers = [1]
117 compute_stats(numbers)
118 # Test case 6: Case with an empty array (length 0)
119 numbers = []
120 compute_stats(numbers)
121 """
122 # Generate and display CFG
123 cfg = create_cfg()
124 draw_cfg(cfg)

```

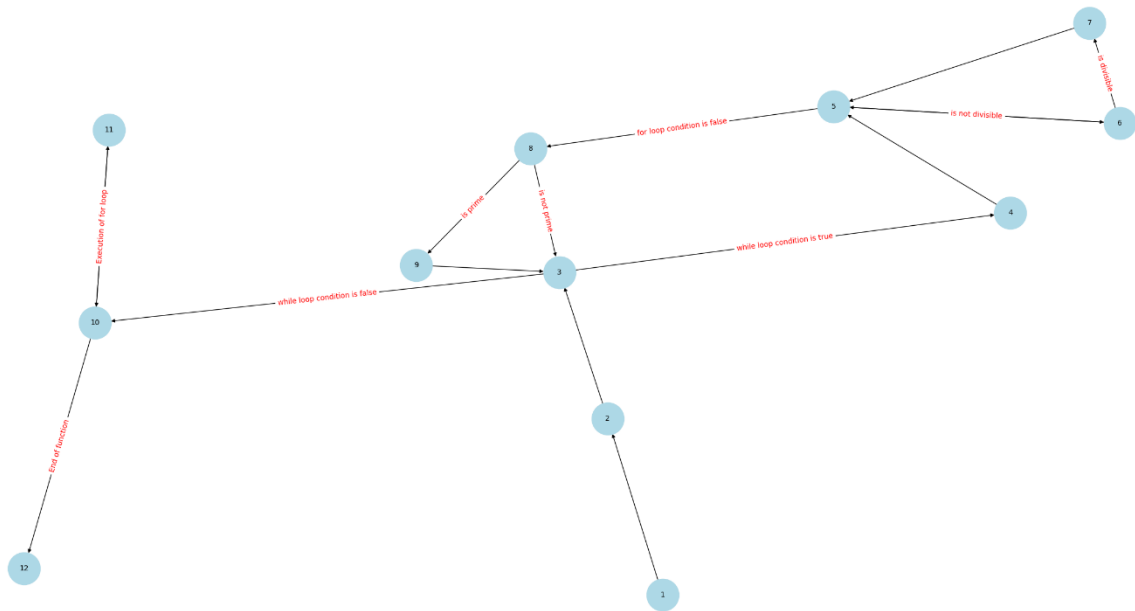
5. Can't run all test cases to cover du paths

- **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: $1 \rightarrow 2 \rightarrow 3 \rightarrow 10 \rightarrow 11 \rightarrow 10 \rightarrow 12$
 - 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: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 5 \rightarrow 8 \rightarrow 9 \rightarrow 3 \rightarrow 10 \rightarrow 11 \rightarrow 10 \rightarrow 12$
 - 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:

```
Q1.py X
Q1.py > edge_coverage_test_cases
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 def power_function(x, y):
5     """
6     Computes the power function  $Z = X^Y$ .
7     Handles both positive and negative exponents.
8     """
9     visited_nodes = []
10
11     visited_nodes.append("1")
12     print(f"Computing {x}^{y}")
13
14     w = abs(y) # Take absolute value of exponent
15     z = 1 # Initialize result
16
17     visited_nodes.append("2")
18     while w != 0:
19         visited_nodes.append("3")
20         z = z * x
21         w = w - 1
22         visited_nodes.append("2")
23     visited_nodes.append("4") # End of loop
24
25     visited_nodes.append("5") # If y
26     if y < 0: # If exponent is negative, take reciprocal
27         visited_nodes.append("6")
28         z = 1 / z
29     visited_nodes.append("7") # End of if
30     print(f"Result: {z}")
31     visited_nodes.append("8") # Print Z and End of function
32
33     print(f"Visited Nodes: {visited_nodes}")
34     return z # Return value for testing
35
36 def create_cfg():
37     """
38     Generates a simplified Control Flow Graph (CFG) for the power function.
39     """
40     cfg = nx.DiGraph()
41
42     # Nodes representing different parts of the program
43     cfg.add_nodes_from([
44         "1", "2", "3", "4", "5", "6", "7", "8"
45     ])
46
47     # Edges representing control flow
48     edges = [
49         ("1", "2"),
50         ("2", "3"), # while true
51         ("2", "4"), # while false, skip loop
52         ("3", "2"), # Loop back
53         ("2", "4"),
54         ("4", "5"),
55         ("5", "6"), # if true
56         ("5", "7"), # if false, skip body of if
57         ("6", "7"),
58         ("7", "8"), # End of function
59     ]
60
61     cfg.add_edges_from(edges)
62     return cfg
```

```

63
64 def draw_cfg(cfg):
65     """
66     Draws the generated CFG using NetworkX and Matplotlib.
67     """
68     pos = nx.spring_layout(cfg) # Position nodes
69     plt.figure(figsize=(10, 6))
70     nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)
71
72     # Define edge labels
73     edge_labels = {
74         ("2", "3"): "while true",
75         ("2", "4"): "while false",
76         ("5", "6"): "if true",
77         ("5", "7"): "if false"
78     }
79
80     # Draw edge labels
81     nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')
82
83     plt.title("Control Flow Graph of Power Function")
84     plt.show()
85
86 def identify_infeasible_paths():
87     # In this program, there are no infeasible paths
88     infeasible_paths = []
89     return infeasible_paths
90
91 def node_coverage_test_cases():
92     # Test cases for node coverage
93     test_cases = [
94         {'X': 2, 'Y': -3}, # Negative Y path travels to all nodes
95     ]
96     return test_cases
97
98 def edge_coverage_test_cases():
99     # Test cases for edge coverage
100     test_cases = [
101         {'X': 2, 'Y': 3}, # positive Y
102         {'X': 2, 'Y': 0} # zero Y
103     ]
104     return test_cases
105
106 def run_test_cases(test_cases):
107     for test in test_cases:
108         print(f"Test case: {test}")
109         power_function(test['X'], test['Y'])
110
111 if __name__ == "__main__":
112     # Generate and display CFG
113     cfg = create_cfg()
114     draw_cfg(cfg)
115
116     # Identify infeasible paths
117     print("\nInfeasible Paths:", identify_infeasible_paths())
118
119     # Provide test cases for node coverage
120     print("\nNode Coverage Test Cases:", node_coverage_test_cases())
121     run_test_cases(node_coverage_test_cases())
122
123     # Provide test cases for edge coverage
124     print("\nEdge Coverage Test Cases:", edge_coverage_test_cases())
125     run_test_cases(edge_coverage_test_cases())

```

Q2:

```
Q2.py X
Q2.py > is_palindrome
1 import networkx as nx
2 import matplotlib.pyplot as plt
3
4 def is_palindrome(s):
5     """
6     Checks if the given string is a palindrome.
7     """
8     visited_nodes = []
9
10    visited_nodes.append("1") #Start
11    visited_nodes.append("2") #if s is None
12    if s is None:
13        visited_nodes.append("3") #throw NullPointerException
14        raise NullPointerException(f"Visited Nodes: {visited_nodes}\n")
15    else:
16        visited_nodes.append("4")
17        left = 0
18        right = len(s) - 1
19        result = True
20
21    visited_nodes.append("5") #while loop
22    while left < right and result:
23        visited_nodes.append("6") #in while loop, executing if condition
24        if s[left] != s[right]:
25            visited_nodes.append("7") #if condition is true
26            result = False
27
28        visited_nodes.append("8") #left and right are decremented
29        left += 1
30        right -= 1
31
32    visited_nodes.append("9") # Check the while loop condition again
33
34    visited_nodes.append("9") #return result
35    print(f"Visited Nodes: {visited_nodes}\n")
36
37    return result
38
39
40 class NullPointerException(Exception):
41     pass
42
43 def create_cfg():
44     """
45     Generates a Control Flow Graph (CFG) for the is_palindrome function.
46     """
47     cfg = nx.DiGraph()
48
49     # Nodes representing different parts of the program
50     cfg.add_nodes_from([
51         "1", "2", "3", "4", "5", "6", "7", "8", "9"
52     ])
53
54     # Edges representing control flow
55     edges = [
56         ("1", "2"),
57         ("2", "3"), # if s is None
58         ("2", "4"), # else
59         ("4", "5"),
60         ("5", "6"), # while true
61         ("5", "9"), # while false
62         ("6", "7"), # if true
63         ("6", "8"), # if false
64         ("7", "8"),
65         ("8", "5") # loop back
66     ]
67
```

```

67
68     cfg.add_edges_from(edges)
69     return cfg
70
71 def draw_cfg(cfg):
72     """
73     Draws the generated CFG using NetworkX and Matplotlib.
74     """
75     pos = nx.spring_layout(cfg) # Position nodes
76     plt.figure(figsize=(10, 6))
77     nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)
78
79     # Define edge labels
80     edge_labels = {
81         ("2", "3"): "if s is None",
82         ("2", "4"): "else",
83         ("5", "6"): "while true",
84         ("6", "7"): "if true",
85         ("6", "8"): "if false",
86         ("8", "5"): "loop back",
87         ("5", "9"): "while false"
88     }
89
90     # Draw edge labels
91     nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')
92
93     plt.title("Control Flow Graph of is_palindrome Function")
94     plt.show()
95
96 def node_coverage_test_cases():
97     # Test cases for node coverage
98     test_cases = [
99         {'s': None}, # throws NullPointerException
100         {'s': "hello"} # not a palindrome
101     ]
102
103     return test_cases
104
105 def edge_coverage_test_cases():
106     # Test cases for edge coverage
107     test_cases = [
108         {'s': "racecar"}, # palindrome
109         {'s': "he"}, # not a palindrome
110         {'s': None} # throws NullPointerException
111     ]
112
113     return test_cases
114
115 def edge_pair_coverage_test_cases():
116     # Test cases for edge-pair coverage
117     test_cases = [
118         {'s': "racecar"}, # palindrome
119         {'s': "abfhba"}, # not a palindrome
120         {'s': "aa"}, # even length palindrome
121         {'s': "ba"}, # even length not a palindrome
122         {'s': None}, # throws NullPointerException
123         {'s': "G"}, # single letter palindrome
124     ]
125
126     return test_cases

```

```

125
126 def prime_path_coverage_test_cases():
127     # Test cases for prime path coverage
128     test_cases = [
129         {'s': "racecar"}, # palindrome
130         {'s': "abcxba"}, # even long length not a palindrome
131         {'s': None},      # throws NullPointerException
132         {'s': "G"},       # single letter palindrome
133         {'s': "ba"}       # even length not a palindrome
134     ]
135     return test_cases
136
137 def run_test_cases(test_cases):
138     for test in test_cases:
139         print(f"Test case: {test}")
140         try:
141             is_palindrome(test['s'])
142         except Exception as e:
143             print(f"Exception: {e}")
144
145 if __name__ == "__main__":
146     # Generate and display CFG
147     cfg = create_cfg()
148     draw_cfg(cfg)
149
150     # Provide test cases for node coverage
151     print("\nNode Coverage Test Cases:", node_coverage_test_cases())
152     run_test_cases(node_coverage_test_cases())
153
154
155     # Provide test cases for edge coverage
156     print("\nEdge Coverage Test Cases:", edge_coverage_test_cases())
157     run_test_cases(edge_coverage_test_cases())
158
159     # Provide test cases for edge-pair coverage
160     print("\nEdge-Pair Coverage Test Cases:", edge_pair_coverage_test_cases())
161     run_test_cases(edge_pair_coverage_test_cases())
162
163     # Provide test cases for prime path coverage
164     print("\nPrime Path Coverage Test Cases:", prime_path_coverage_test_cases())
165     run_test_cases(prime_path_coverage_test_cases())
166

```

Q2_test.py X

Q2_test.py > TestIsPalindrome > test_palindrome

```
1  import unittest
2
3  from Q2 import *
4
5  class TestIsPalindrome(unittest.TestCase):
6
7      def test_palindrome(self):
8          self.assertTrue(is_palindrome("racecar"))
9
10     def test_even_length_not_palindrome(self):
11         self.assertFalse(is_palindrome("abcxba"))
12
13     def test_null_string(self):
14         with self.assertRaises(NullPointerException):
15             is_palindrome(None)
16
17     def test_single_letter_palindrome(self):
18         self.assertTrue(is_palindrome("G"))
19
20     def test_even_length_not_palindrome_2(self):
21         self.assertFalse(is_palindrome("ba"))
22
23     if __name__ == "__main__":
24         # Generate and display CFG
25         cfg = create_cfg()
26         draw_cfg(cfg)
27
28         # Run the test cases for prime path coverage
29         print("\nRunning Prime Path Coverage Test Cases:")
30         unittest.main(argv=[''], verbosity=2, exit=False)
```

Q3:

```
Q3.py x
Q3.py > compute_stats
1 import math
2 import networkx as nx
3 import matplotlib.pyplot as plt
4
5 def compute_stats(numbers):
6     """
7     Computes Mathematical Functions given a set of numbers.
8     """
9     visited_nodes = []
10    visited_nodes.append("1") # Start
11    visited_nodes.append("2") # Initialize variables
12    length = len(numbers)
13    sum_values = 0
14
15    # Loop to calculate the sum
16    visited_nodes.append("3") # Loop to calculate the sum
17    for i in range(length):
18        visited_nodes.append("4") # sum_values += numbers[i]
19        sum_values += numbers[i]
20        visited_nodes.append("3")
21
22    visited_nodes.append("5") # Sort the numbers to find the median
23    # Sort the numbers to find the median
24    numbers_sorted = sorted(numbers)
25    median = numbers_sorted[length // 2]
26    # Calculate the mean
27    mean = sum_values / length
28    varsum = 0
29
30    # Loop to calculate the variance sum
31    visited_nodes.append("6") # Loop to calculate the variance sum
32    for i in range(length):
33        visited_nodes.append("7")
34        varsum += (numbers[i] - mean) ** 2
35        visited_nodes.append("6")
36
37    visited_nodes.append("8") # Calculate the variance
38    # Calculate the variance
39    variance = varsum / (length - 1)
40    # Calculate the standard deviation
41    standard_deviation = math.sqrt(variance)
42    # Print the results
43    print(f"length: {length}")
44    print(f"mean: {mean}")
45    print(f"median: {median}")
46    print(f"variance: {variance}")
47    print(f"standard deviation: {standard_deviation}")
48    print(f"Visited Nodes: {visited_nodes}\n")
49
```



```
50 def create_cfg():
51     """
52     Generates a Control Flow Graph (CFG) for the compute_stats function.
53     """
54     cfg = nx.DiGraph()
55
56     # Nodes representing different parts of the program
57     cfg.add_nodes_from([
58         "1", "2", "3", "4", "5", "6", "7", "8"
59     ])
60
61     # Edges representing control flow
62     edges = [
63         ("1", "2"),
64         ("2", "3"),
65         ("3", "4"),
66         ("4", "3"), # Loop back
67         ("3", "5"),
68         ("5", "6"),
69         ("6", "7"),
70         ("7", "6"), # Loop back
71         ("6", "8")
72     ]
73
74     cfg.add_edges_from(edges)
75     return cfg
```

```

77 def draw_cfg(cfg):
78     """
79     Draws the generated CFG using NetworkX and Matplotlib.
80     """
81     pos = nx.spring_layout(cfg) # Position nodes
82     plt.figure(figsize=(10, 6))
83     nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)
84
85     # Define edge labels
86     edge_labels = {
87         ("4", "3"): "Loop through numbers",
88         ("3", "5"): "Exit for loop",
89         ("7", "6"): "Loop through numbers",
90         ("6", "8"): "Exit for loop"
91     }
92
93     # Draw edge labels
94     nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')
95
96     plt.title("Control Flow Graph of compute_stats Function")
97     plt.show()
98
99     # Test case 1: Normal case with positive numbers
100    numbers = [1, 2, 3, 4, 5]
101    compute_stats(numbers)
102
103    # Test case 2: Case with negative numbers
104    numbers = [-1, -2, -3, -4, -5]
105    compute_stats(numbers)
106
107    # Test case 3: Case with mixed positive and negative numbers
108    numbers = [-1, 2, -3, 4, -5]
109    compute_stats(numbers)
110
111    # Test case 4: Case with all zeros
112    numbers = [0, 0, 0, 0, 0]
113    compute_stats(numbers)
114    """
115    # Test case 5: Case with a single element
116    numbers = [1]
117    compute_stats(numbers)
118    # Test case 6: Case with an empty array (length 0)
119    numbers = []
120    compute_stats(numbers)
121    """
122    # Generate and display CFG
123    cfg = create_cfg()
124    draw_cfg(cfg)

```

Q4:

```
Q4.py  X
Q4.py > print_primes
1  import math
2  import networkx as nx
3  import matplotlib.pyplot as plt
4
5  def is_divisible(a, b):
6      return b % a == 0
7
8  def print_primes(n):
9      """
10     Finds and prints n prime integers.
11     """
12     visited_nodes = [] # visited nodes
13     visited_nodes.append(1) # Start
14     visited_nodes.append(2) # Execution of initialization
15     cur_prime = 2 # Value currently considered for primeness
16     num_primes = 1 # Number of primes found so far
17     primes = [0] * 100 # The list of prime numbers
18     primes[0] = 2 # Initialize 2 into the list of primes
19
20     visited_nodes.append(3) # Check Condition of while loop
21     while num_primes < n:
22         visited_nodes.append(4) # Execution of while loop
23         cur_prime += 1 # Next number to consider
24         is_prime = True
25
26         visited_nodes.append(5) # Check Condition of for loop
27         for i in range(num_primes): # For each previous prime
28             visited_nodes.append(6) # Check Condition of if statement
29             if is_divisible(primes[i], cur_prime): # Found a divisor, cur_prime is not prime
30                 visited_nodes.append(7) # Execution of if statement
31                 is_prime = False
32                 break # Out of loop through primes
33             visited_nodes.append(5) # Check condition of for loop
34
35         visited_nodes.append(8) # Check Condition of if statement
36         if is_prime: # Save it
37             visited_nodes.append(9) # Execution of if statement
38             primes[num_primes] = cur_prime
39             num_primes += 1
40
41     visited_nodes.append(3) # Check Condition of while loop
42
43     # Print all the primes out
44     visited_nodes.append(10) # Execution of for loop
45     for i in range(num_primes):
46         visited_nodes.append(11)
47         print(f"Prime: {primes[i]}")
48         visited_nodes.append(10) # Check Condition of for loop
49
50     visited_nodes.append(12) # End of function
51     print(f"Visited Nodes: {visited_nodes}\n")
```

```

53 def create_cfg():
54     """
55     Generates a Control Flow Graph (CFG) for the print_primes function.
56     """
57     cfg = nx.DiGraph()
58
59     # Nodes representing different parts of the program
60     cfg.add_nodes_from([
61         "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12"
62     ])
63
64     # Edges representing control flow
65     edges = [
66         ("1", "2"),      # Start -> Initialization
67         ("2", "3"),      # Initialization -> Check while loop condition
68         ("3", "4"),      # while loop condition is true
69         ("3", "10"),     # while loop condition is false
70         ("4", "5"),      # Execution of while loop -> Check for loop condition
71         ("5", "6"),      # for loop condition is true
72         ("5", "8"),      # for loop condition is false
73         ("6", "5"),      # is not divisible -> Check for loop condition
74         ("6", "7"),      # is divisible
75         ("7", "5"),      # Loop back to check for loop condition
76         ("8", "9"),      # is prime
77         ("8", "3"),      # is not prime
78         ("9", "3"),      # Loop back to check while loop condition
79         ("10", "11"),    # Execution of for loop -> Print primes
80         ("11", "10"),    # Loop back to check for loop condition
81         ("10", "12")     # End of function
82     ]
83
84     cfg.add_edges_from(edges)
85     return cfg

```

```

87  def draw_cfg(cfg):
88      """
89      Draws the generated CFG using NetworkX and Matplotlib.
90      """
91      pos = nx.spring_layout(cfg) # Position nodes
92      plt.figure(figsize=(10, 6))
93      nx.draw(cfg, pos, with_labels=True, node_color='lightblue', edge_color='black', font_size=10, node_size=2000)
94
95      # Define edge labels
96      edge_labels = {
97          ("3", "4"): "while loop condition is true",
98          ("3", "10"): "while loop condition is false",
99          ("5", "8"): "for loop condition is false",
100         ("6", "5"): "is not divisible",
101         ("6", "7"): "is divisible",
102         ("8", "9"): "is prime",
103         ("8", "3"): "is not prime",
104         ("10", "11"): "Execution of for loop",
105         ("10", "12"): "End of function"
106     }
107
108     # Draw edge labels
109     nx.draw_networkx_edge_labels(cfg, pos, edge_labels=edge_labels, font_color='red')
110
111     plt.title("Control Flow Graph of print_primes Function")
112     plt.show()
113
114     # Example usage
115     print_primes(1)
116
117     # Generate and display CFG
118     cfg = create_cfg()
119     draw_cfg(cfg)

```

```

Q4_test.py X
Q4_test.py > ...
1  import unittest
2  from unittest.mock import patch
3  from io import StringIO
4  from Q4 import print_primes
5
6  class TestPrintPrimes(unittest.TestCase):
7
8      @patch('sys.stdout', new_callable=StringIO)
9      def test_path_1(self, mock_stdout):
10         """
11         Test path: 1 -> 2 -> 3 -> 10 -> 11 -> 10 -> 12
12         Input: n = 1
13         Explanation: This path covers the edges from the start to the initialization,
14         checks the while loop condition (which is false), and then moves to the for loop to print the primes.
15         """
16         expected_output = "Prime: 2\nVisited Nodes: [1, 2, 3, 10, 11, 10, 12]\n"
17         print_primes(1)
18         self.assertEqual(mock_stdout.getvalue().strip(), expected_output.strip())
19
20     @patch('sys.stdout', new_callable=StringIO)
21     def test_path_2(self, mock_stdout):
22         """
23         Test path: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 5 -> 8 -> 9 -> 3 -> 10 -> 11 -> 10 -> 12
24         Input: n = 2
25         Explanation: This path covers the edges from the start to the initialization,
26         checks the while loop condition (which is true), executes the while loop body,
27         checks the for loop condition, finds a prime, and then moves to the for loop to print the primes.
28         """
29         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"
30         print_primes(2)
31         self.assertEqual(mock_stdout.getvalue().strip(), expected_output.strip())
32
33 if __name__ == '__main__':
34     unittest.main()

```