Main Flow Services And Technologies
PYTHON PROGRAMMING INTERNSHIP
Task - 2

9. Prime Number:

```
def is_prime(n): # define prime function
  if n <= 1:
    return False
  for i in range(2, (n//2)+1): # loops till it reaches the condition
    if n % i == 0: # sees that number only divisible by 1 and itself
      return False
  return True
num = int(input("Enter a number: ")) # input from user
is_prime(num) # function call
```

10. Sum of Digits:

```
def sum_digits(n): # define sum of digits function
  sum = 0
  while n > 0: # loops through the digits
    sum += n % 10 # gets the last digit
    n //= 10 # removes the last digits
  return sum # returns the sum
num = int(input("Enter a number: ")) # input from user
sum_digits(num) # function call
```

11. LCM and GCD:

```
import math # import math library
def gcd_lcm(n, m): # define function for gcd and lcm
  gcd = math.gcd(n, m) # math.gcd() inbuilt function is used
  lcm = (n * m) // gcd # product of 2 numbers is floor division by gcd
  return gcd, lcm
num1 = int(input("Enter first number: ")) # input from user
num2 = int(input("Enter second number: ")) # input from user
gcd, lcm = gcd_lcm(num1, num2) # the gcd _lcm() is called
print(f"GCD: {gcd}") # f is used for getting formatted output
```

```
print(f"LCM: {lcm}")
```

12. List Reversal:

```
def reverse_list(list): # define function of reverse list
  list = list[::-1] # list slice operation is done
  return list # the -1 returns the list backwards
lst = list(map(int, input("Enter the elements of the list: ").split()))
# the input of the list integer is got using functions like map and list
reverse_list(lst) # function call
```

13. Sort a List:

```
def sort_num_list(list): # define function
  list = sorted(list) # sorted() is used to sort the list
  return list
lst = list(map(int, input("Enter the elements of the list: ").split()))
# the input of the list integer is got using functions like map and list
sort_num_list(lst) # function call
```

14. Remove Duplicates:

```
def remove_duplicates(l): # define function
  l = list(set(l)) # in set duplicate elements will be removed
  return l # then it is converted to list
lst = list(map(int, input("Enter the elements of the list: ").split()))
# the input of the list integer is got using functions like map and list
remove_duplicates(lst) # function call
```

15. String Length:

```
def count_words(str): # define function
  count = 0 # initialized to 0
  for char in str: # traverses  through the sting
      count += 1 # increments 1
  return count
str = input("Enter a string: ") # input from user
print(count_words(str)) # function call
```

16. Count Vowels and Consonants:

```
def count_vowels(str): # define function of vowel count
  vowels = "aeiouAEIOU" # both lower & upper case
```

```
  count = 0 # initialized to 0
  for char in str: # traverses  through the sting
    if char in vowels: # checks for match with vowel with each char of string
      count += 1 # increments 1
  return count
count_vowels(str)
def count_consonants(str): # define function of consonant count
  consonants = "bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ"
# both lower & upper case
  count = 0 # initialized to 0
  for char in str: # traverses  through the sting
    if char in consonants:
 # checks for match with consonants with each char of string
      count += 1 # increments 1
  return count
str  = input("Enter a string: ")
vow = count_vowels(str) # function call
cons = count_consonants(str) # function call
print(f"Number of vowels: {vow}")  # f is used for getting formatted output
print(f"Number of consonants: {cons}") # f is used to get formatted output
```

2. Maze Generator and Solver:

```
import random  # importing the random module for shuffling directions
def generate_maze(width, height): # function to generate a random maze
    maze = [[1 for _ in range(width)] for _ in range(height)]
    # creating a 2D grid filled with walls, represented by 1
    def carve_passages(x, y):  # nested func to carve out paths in the maze
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        # defining possible directions to move (down, right, up, left)
        random.shuffle(directions)  # shuffling directions ensure randomness
        for dx, dy in directions:  # loop through each direction
            nx, ny = x + dx * 2, y + dy * 2  # calculate next , skipping one cell
            if 0 <= nx < width and 0 <= ny < height and maze[ny][nx] == 1:
                # check if the next cell is within bounds and still a wall
                maze[y + dy][x + dx] = 0  # remove the wall current and next cell
                maze[ny][nx] = 0  # mark the next cell as a path (0)
                carve_passages(nx, ny)  # recursively carve path for the next cell
```

```
    maze[1][1] = 0  # set the starting point as a path
    carve_passages(1, 1)  # start carving passages from the starting point
    maze[0][1] = 0  # open an entry point at the top of the maze
    maze[height - 1][width - 2] = 0  # open exit point at the bottom of maze
    return maze  # return the completed maze
def solve_maze(maze, start, end):  # function to solve the maze using DFS
    stack = [start]  # stack to keep track of the cells to visit
    visited = set()  # set to track visited cells
    path = {}  # dictionary to store the path (key = cell, value = previous cell)
    while stack:  # loop until the stack is empty
        x, y = stack.pop()  # take the last cell from the stack
        if (x, y) == end:  # if the current cell is the end, stop the search
            break
        if (x, y) not in visited:  # if the cell is not yet visited
            visited.add((x, y))  # mark it as visited
            for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
                # loop through possible directions (down, right, up, left)
                nx, ny = x + dx, y + dy  # calc the coordinates of the next cell
                if (0 <= nx < len(maze[0]) and 0 <= ny < len(maze) and
                        maze[ny][nx] == 0 and (nx, ny) not in visited):
                    # check if the next cell is within bounds, a path, and not visited
                    stack.append((nx, ny))  # add the next cell to the stack
                    path[(nx, ny)] = (x, y)  # record the current cell as the
predecessor
    solution_path = []  # list to store the solution path
    cell = end  # start from the end cell
    while cell != start:  # trace the path back to the start
        solution_path.append(cell)  # add the current cell to the path
        cell = path[cell]  # move to the predecessor cell
    solution_path.append(start)  # add the start cell to the path
    solution_path.reverse()  # reverse the path to start from the beginning
    return solution_path  # return the solution path
def display_maze(maze, path=None):  # function to display the maze
width = len(maze[0])  # get the width of the maze
    height = len(maze)  # get the height of the maze
    print("+" + "-" * (2 * width - 1) + "+")  # print the top border of the maze
    for y in range(height):  # loop through each row of the maze
```

```
        row = "|"  # start the row with a left border
        for x in range(width):  # loop through each cell in the row
            if path and (x, y) in path:  # if the cell is part of the solution path
                row += " ."  # mark it with a dot
            else:
                row += "  " if maze[y][x] == 0 else " #"
                # print a space for paths (0) and a hash for walls (1)
        row += "|"  # end the row with a right border
        print(row)  # print the current row
    print("+" + "-" * (2 * width - 1) + "+")  # print the bottom border of maze
maze_width = 11  # define the width of the maze
maze_height = 11  # define the height of the maze
maze = generate_maze(maze_width, maze_height)  # gen random maze
print("Generated Maze:")  # message indicating the initial maze
display_maze(maze)  # display the generated maze without a solution path
start = (1, 0)  # set the starting point
end = (maze_width - 2, maze_height - 1)  # set the ending point
solution_path = solve_maze(maze, start, end)
# solve the maze to find the solution path
print("\nSolved Maze:")  # message indicating the solved maze
display_maze(maze, solution_path)  # display the maze with solution path
```