# TASK 2

**9. Prime Number**: Determine if a number is prime.

**Code:**

```
def is_prime(n):
  if n <= 1:
    return False  # Not prime
  for i in range(2, int(n**0.5) + 1):  # Check divisors up to √n
    if n % i == 0:
      return False  # Divisible by i, not prime
  return True


n = int(input("Enter a number to check if it is prime: "))
print(f"Is {n} a prime number? {is_prime(n)}")
```

**Explanation:**

- Prime numbers are greater than 1 and divisible only by 1 and themselves.
- To check if a number is prime:
    - If n≤1, return False.
    - Iterate from 2 to √n and check if n%i==0. If true, it's not prime.
    - If no divisors are found, return True.

---

**10. Sum of Digits**: Find the sum of digits in a number.

**Code:**

```
def sum_of_digits(n):
  return sum(int(digit) for digit in str(abs(n)))
n = int(input("Enter a number to find the sum of its digits: "))
print(f"The sum of digits in {n} is {sum_of_digits(n)}")
```

**Explanation:**

- Convert the number to a string to access each digit individually.
- Use a loop or comprehension to convert each digit back to an integer and calculate the sum.
- Use abs() to handle negative numbers.

---

**11. LCM and GCD**: Calculate LCM and GCD of two integers.

**Code:**

```
import math

def lcm_and_gcd(a, b):
    gcd = math.gcd(a, b)  # Built-in GCD
    lcm = abs(a * b) // gcd  # LCM formula: |a * b| / GCD
    return lcm, gcd

a = int(input("Enter the first number: "))

b = int(input("Enter the second number: "))

lcm, gcd = lcm_and_gcd(a, b)

print(f"The LCM of {a} and {b} is {lcm}, and the GCD is {gcd}")
```

**Explanation:**

- GCD is the largest number that divides both numbers.
- LCM is calculated using LCM=|a·b|/GCD

---

**12. List Reversal**: Reverse a list without built-in functions.

**Code:**

```
def reverse_list(lst):
    reversed_list = []
    for i in range(len(lst) - 1, -1, -1):
        reversed_list.append(lst[i])
    return reversed_list
```

```
lst = list(map(int, input("Enter a list of integers separated by spaces: ").split()))
print(f"The reversed list is: {reverse_list(lst)}")
```

**Explanation:**

- Use a loop to iterate through the list in reverse order.
- Append each element to a new list.

---

**13. Sort a List**: Sort a list of numbers in ascending order.

**Code:**

```
def bubble_sort(lst):
    n = len(lst)
    for i in range(n - 1):
        for j in range(n - i - 1):  # Compare adjacent elements
            if lst[j] > lst[j + 1]:
                lst[j], lst[j + 1] = lst[j + 1], lst[j]  # Swap if out of order
    return lst
lst = list(map(int, input("Enter a list of integers separated by spaces to sort: ").split()))
print(f"The sorted list is: {bubble_sort(lst)}")
```

**Explanation:**

- Bubble Sort compares adjacent elements and swaps them if they’re in the wrong order.
- Repeat the process until the entire list is sorted.

---

**14. Remove Duplicates**: Remove duplicate elements from a list.

**Code:**

```
def remove_duplicates(lst):
```

```
    unique_list = []
    for item in lst:
        if item not in unique_list:  # Add only if not already in unique_list
            unique_list.append(item)
    return unique_list

lst = list(map(int, input("Enter a list of integers separated by spaces: ").split()))
print(f"The list without duplicates is: {remove_duplicates(lst)}")
```

**Explanation:**

- Use a loop to check if an element is already in the result list.
- If not, add it.

---

**15. String Length**: Find the length of a string without len().

**Code:**

```
def string_length(s):
    count = 0
    for char in s:  # Iterate through each character
        count += 1
    return count

s = input("Enter a string to find its length: ")
print(f"The length of the string is: {string_length(s)}")
```

**Explanation:**

- Use a counter to count each character in the string.

---

**16. Count Vowels and Consonants**: Count vowels and consonants in a string.

**Code:**

```
def count_vowels_and_consonants(s):
  vowels = set('aeiouAEIOU')
  v_count = c_count = 0
  for char in s:
    if char.isalpha():  # Check if character is a letter
      if char in vowels:
        v_count += 1  # Count vowel
      else:
        c_count += 1  # Count consonant
  return v_count, c_count


s = input("Enter a string to count vowels and consonants: ")
vowels, consonants = count_vowels_and_consonants(s)
print(f"The number of vowels is: {vowels}")
print(f"The number of consonants is: {consonants}")
```

**Explanation:**

- Use a set to define vowels.
- Loop through each character and check:
    - If it's a vowel, increment the vowel count.
    - If it's a consonant, increment the consonant count.

---

**2. Maze Generator and Solver**: Generate and solve mazes using DFS or BFS.

**Code :**

```
import random


def generate_maze(rows, cols):
```

```
    maze = [[1] * cols for _ in range(rows)]  # Initialize maze with walls (1)

    def dfs(x, y):
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up
        random.shuffle(directions)  # Randomize directions for variety
        maze[x][y] = 0  # Mark current cell as a path (0)

        for dx, dy in directions:
            nx, ny = x + dx * 2, y + dy * 2  # Look two steps in the chosen direction
            if 1 <= nx < rows - 1 and 1 <= ny < cols - 1 and maze[nx][ny] == 1:
                maze[x + dx][y + dy] = 0  # Remove the wall between cells
                dfs(nx, ny)  # Recursively carve paths

    dfs(1, 1)  # Start carving paths from (1, 1)
    return maze

def print_maze(maze):
    for row in maze:
        print("".join("█" if cell == 1 else " " for cell in row))  # Walls as █, paths as space

def solve_maze(maze):
    rows, cols = len(maze), len(maze[0])
    start, end = (1, 1), (rows - 2, cols - 2)  # Start and end positions
    stack = [start]  # Stack for DFS
    visited = set()  # Keep track of visited cells
    path = {}  # To reconstruct the solution path

    while stack:
```

```
        x, y = stack.pop()
        if (x, y) == end:  # Reached the end
            solution_path = []
            while (x, y) != start:
                solution_path.append((x, y))
                x, y = path[(x, y)]
            solution_path.append(start)
            return solution_path[::-1]  # Reverse the path for correct order

        visited.add((x, y))  # Mark the cell as visited
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:  # Right, Down, Left, Up
            nx, ny = x + dx, y + dy
            if (
                0 <= nx < rows and 0 <= ny < cols  # Within bounds
                and maze[nx][ny] == 0  # Path
                and (nx, ny) not in visited  # Not visited
            ):
                stack.append((nx, ny))  # Add next cell to stack
                path[(nx, ny)] = (x, y)  # Record the path

    return []  # No solution found

def display_solution(maze, solution_path):
    solved_maze = [row[:] for row in maze]  # Copy the maze
    for x, y in solution_path:
        solved_maze[x][y] = "."  # Mark the solution path with dots
    for row in solved_maze:
        print("".join("" if cell == 1 else "." if cell == "." else " " for cell in row))
```

```
# Interactive Test
rows = int(input("Enter the number of rows for the maze (odd number): "))
cols = int(input("Enter the number of columns for the maze (odd number): "))

maze = generate_maze(rows, cols)
print("\nGenerated Maze:")
print_maze(maze)

solution = solve_maze(maze)
if solution:
    print("\nSolved Maze (Path shown as '.'): ")
    display_solution(maze, solution)
else:
    print("\nNo solution found for this maze.")
```

**Explanation:**

- **Maze Generation:**
    - Use Depth-First Search (DFS) to carve paths in a grid.
    - Mark cells as walls (1) and paths (0).
- **Maze Solver:**
    - Use a stack to explore paths from the start to the end.
    - Mark visited cells to avoid cycles.