```python
# Display the environment
def display_environment(grid, vacuum_pos):
    """
    Display the current environment.
    Args:
        grid (list[list[str]]): The environment grid.
        vacuum_pos (tuple[int, int]): The position of the vacuum cleaner.
    """
    for i, row in enumerate(grid):
        for j, cell in enumerate(row):
            if (i, j) == vacuum_pos:
                print('V', end=' ')  # Represent the vacuum cleaner
            else:
                print(cell, end=' ')
        print()
    print()

# Get the initial environment from the user
def get_user_environment(rows, cols):
    """
    Get the environment setup from the user.
    Args:
        rows (int): Number of rows in the grid.
        cols (int): Number of columns in the grid.

    Returns:
        list[list[str]]: The user-defined environment grid.
    """
    grid = []
    print("Enter the grid status row by row (C for Clean, D for Dirty):")
```

```python
for i in range(rows):
    while True:
        row_input = input(f"Row {i + 1} (e.g., C D D C): ").split()
        if len(row_input) == cols and all(cell in ['C', 'D'] for cell in row_input):
            grid.append(row_input)
            break
        else:
            print(f"Invalid input. Enter exactly {cols} values, each being 'C' or 'D'.")
return grid

cuum cleaner agent
vacuum_cleaner_agent(grid, start_pos):
"""
Simulate the vacuum cleaner agent cleaning the environment autonomously.
Args:
    grid (list[list[str]]): The environment grid.
    start_pos (tuple[int, int]): Starting position of the vacuum cleaner.

Returns:
    None
"""
rows, cols = len(grid), len(grid[0])
vacuum_pos = start_pos
cleaned_count = 0
total_dirty = sum(row.count('D') for row in grid)

print("\nInitial Environment:")
display_environment(grid, vacuum_pos)

while total_dirty > 0:
```

```python
while total_dirty > 0:
    x, y = vacuum_pos
    # Clean current position if dirty
    if grid[x][y] == 'D':
        grid[x][y] = 'C'
        cleaned_count += 1
        total_dirty -= 1
        print(f"Cleaned position ({x}, {y}).")
    else:
        print(f"Position ({x}, {y}) is already clean.")

    # Display the updated environment
    display_environment(grid, vacuum_pos)

    # Move to the next position systematically
    if y + 1 < cols:  # Move right if possible
        vacuum_pos = (x, y + 1)
    elif x + 1 < rows:  # Move to the next row if possible
        vacuum_pos = (x + 1, 0)
    else:
        break  # No more positions to move to (all cells visited)

print("Cleaning completed!")
display_environment(grid, vacuum_pos)
print(f"Total cleaned: {cleaned_count}")
```

```python
# Main function to run the simulation
if __name__ == "__main__":
    print("Name : Vyom Gupta")
    print("USN  : 1BM22CS333\n")

    rows = int(input("Enter the number of rows in the grid: "))
    cols = int(input("Enter the number of columns in the grid: "))
    environment = get_user_environment(rows, cols)
    start_pos = (0, 0)  # Starting position of the vacuum cleaner
    vacuum_cleaner_agent(environment, start_pos)
```

```
Name  : Vyom Gupta
USN   : 1BM22CS333

Enter the number of rows in the grid: 2
Enter the number of columns in the grid: 3
Enter the grid status row by row (C for Clean, D for Dirty):
Row 1 (e.g., C D D C): D D C
Row 2 (e.g., C D D C): C D D

Initial Environment:
V D C
C D D

Cleaned position (0, 0).
V D C
C D D

Cleaned position (0, 1).
C V C
C D D

Position (0, 2) is already clean.
C C V
C D D

Position (1, 0) is already clean.
C C C
V D D

Cleaned position (1, 1).
C C C
C V D
```

```
Cleaned position (1, 2).
C C C
C C V

Cleaning completed!
C C C
C C V

Total cleaned: 4
```

8 Puzzle Problem Using BFS and DFS :

```python
from collections import deque

# Utility function to display a state
def display_state(state):
    for row in state:
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

# Find the position of the blank (0) in the puzzle
def find_blank(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

# Check if the current state is the goal state
def is_goal(state):
    return state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

# Generate possible moves from the current state
def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]  # Deep copy the state
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
```

```python
# BFS implementation
def bfs(initial_state):
    print("\n--- BFS Solution ---")
    queue = deque([initial_state])
    visited = set()
    visited.add(tuple(tuple(row) for row in initial_state))
    parent = {tuple(tuple(row) for row in initial_state): None}

    while queue:
        current = queue.popleft()

        if is_goal(current):
            path = []
            while current is not None:
                path.append(current)
                current = parent[tuple(tuple(row) for row in current)]
            return path[::-1]  # Return the path from start to goal

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                visited.add(neighbor_tuple)
                parent[neighbor_tuple] = current
                queue.append(neighbor)

    return None  # No solution found
```

```python
# DFS implementation
def dfs(initial_state):
    print("\n--- DFS Solution ---")
    stack = [initial_state]
    visited = set()
    visited.add(tuple(tuple(row) for row in initial_state))
    parent = {tuple(tuple(row) for row in initial_state): None}

    while stack:
        current = stack.pop()

        if is_goal(current):
            path = []
            while current is not None:
                path.append(current)
                current = parent[tuple(tuple(row) for row in current)]
            return path[::-1]  # Return the path from start to goal

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                visited.add(neighbor_tuple)
                parent[neighbor_tuple] = current
                stack.append(neighbor)

    return None  # No solution found
```

```python
print("Name : Vyom Gupta")
print("USN  : 1BM22CS333\n")

print("Enter the initial state of the 8-puzzle (row by row, use 0 for blank):")
initial_state = []
for i in range(3):
    row = list(map(int, input(f"Row {i + 1}: ").split()))
    initial_state.append(row)

print("\nInitial State:")
display_state(initial_state)

# Solve using BFS
bfs_solution = bfs(initial_state)
if bfs_solution:
    print("BFS Solution Steps:")
    for step in bfs_solution:
        display_state(step)
else:
    print("No solution found using BFS.")

# Solve using DFS
dfs_solution = dfs(initial_state)
if dfs_solution:
    print("DFS Solution Steps:")
    for step in dfs_solution:
        display_state(step)
else:
    print("No solution found using DFS.")
```

```
Name : Vyom Gupta
USN  : 1BM22CS333

Enter the initial state of the 8-puzzle (row by row, use 0 for blank):
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 7 0 8

Initial State:
1 2 3
4 5 6
7   8


--- BFS Solution ---
BFS Solution Steps:
1 2 3
4 5 6
7   8

1 2 3
4 5 6
7 8
```

```
--- DFS Solution ---
DFS Solution Steps:
1 2 3
4 5 6
7   8

1 2 3
4 5 6
7 8
```