Project Name: Conway s Game of Life

Team:

ID	Name
20220492	معاذ خالد يحيي ابراهيم
20220096	آية محمد عبد الحليم محمود الشيخ
20220464	مريم سيد فتحي
20220108	بسملة حسام ماهر
20220495	معتز ممدوح حسن
20220486	مصطفي محمود حسني
20220442	محمود صبحي عيد محمود

First, The imperative code

in this block:

```
InitialGrid = []
for i in range(rows):
    row = []
    for j in range(columns):
        row.append(random.randint(0, 1))
    InitialGrid.append(row)
```

we apply:

1 - List construction

2- State mutation (Change the list contents by append)

3- Loops

In this block:

```
    if InitialGrid[row][column] == 1:
    if neighbors < 2:
```

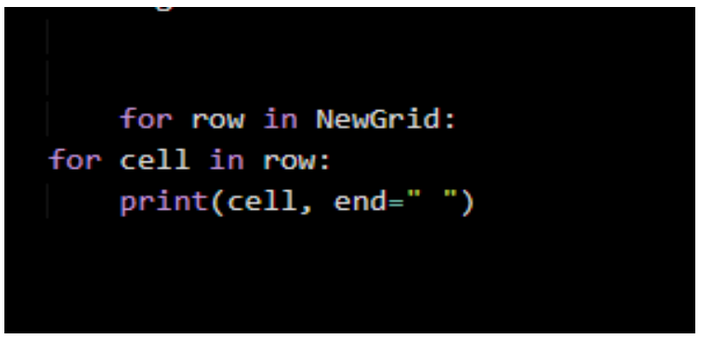
We apply:

1- Conditional branching (Section 4)

2- Imperative State Updates (Section 1)

3- Invariant idea (Section 3)

In this block:



We apply:

List traversal (list of lists) (Section 4)

Throughout the entire code

1 - Imperative paradigm

2 - State manipulation

3 - Sequential execution

4 - Invariant programming

5 - Loops

Functional 1:

```
def sum_tail(list):
def helper(remaining_elements, acc):
    if not remaining_elements:
        return acc
    return helper(remaining_elements[1:], acc + remaining_elements[0])
return helper(list, 0)
```

We apply:

1- Tail recursion (Section 3)

2- Recursive function matching recursive data (Section 4)

3- Accumulator pattern (Section 3)

4- Higher-order idea (function inside function) (Section 5)

In these blocks:

```
def render_row_tail(row):
def helper(remaining_cells, acc):
    if not remaining_cells:
        return acc
```

```
return tuple(tuple(rng.randint(0, 1) for _ in range(cols)) for _ in range(rows))
```

```
return helper(remaining_rows[1:], acc + sum_tail(remaining_rows[0]))
```

We apply:

1- Building strings using recursion (recursive structure like lists)

2- Closure (inner function uses outer variables) (Section 5)

3- Tail recursion

4- Immutable data structure (Section 5)

5- Declarative-like comprehension

6- No mutation

7- Recursion over grid

8- Calling another recursive function [sum_tail] ((higher order use))

9- Accumulator logic

Throughout the entire code

1 - Tail recursion - Accumulator patterns – Recursion – Closures - contextual environment (inner helper accessing outer args)

Functional 2:

In this block:

```
    return sum(
    grid[r + dr][c + dc]
    for dr, dc in deltas
    if 0 <= r + dr < rows and 0 <= c + dc < cols
)
```

We apply:

1-Pure function

2-Declarative rule application

3- Higher-order style //sum takes the function like iterable

5- No state mutation

In this block

```
def next_generation(grid: Grid) -> Grid:
    rows, cols = len(grid), len(grid[0])
    return tuple(
    tuple(
        apply_rule(grid[r][c], count_neighbors(grid, r, c))
        for c in range(cols)
    )
    for r in range(rows)
    )
```

We apply:

1- Nested functional transformations

2- **Functions as transformations** [apply_rule + count_neighbors] it's the (higher order flow) {Section 5}

Throughout the entire code

1 - immutable tuples

2 - pure functions

3 -recursive data shape (grid = rows: each row is a list of cells) but without the actual recursion

4 - functional transformation