```
begin
    using PlutoUI, Colors
end
```

Tile Scrambler

The purpose of this notebook is to create a utility that takes a grid of tiles and permutes it based on the tile flipping rules of Flippin.

Steps

Before doing anything, select a difficulty level for the puzzle. There are 3 difficulties configured.

Difficulty	Description
Easy	5 moves & tiles are flipped once per move
Medium	7 moves & tiles are flipped once per move
Hard	9 moves & tiles are flipped once or twice per move

Easy Confirm

Start with a 25-character string representing the base state of the grid. Possible values are: 0, 1, or 2.

This string will then be broken down into the individual tile values and rearranged into a more useful form for processing.

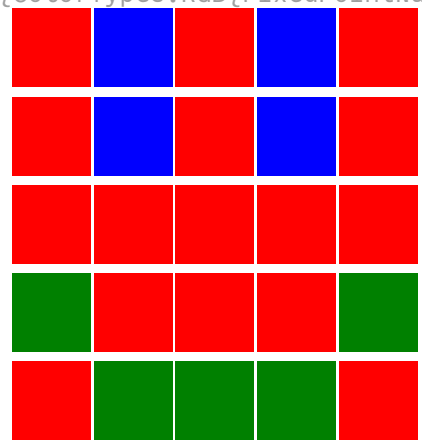
0202002020000001000101110 Confirm

```
basevector = [0, 2, 0, 2, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0]
```

```
base =
[[0, 2, 0, 2, 0], [0, 2, 0, 2, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 1], [0, 1, 1, 1, 0]]
```

This is what the string looks like as a grid of colored tiles.
(You may need to expand the output so that it looks like a grid.)

```
▼Vector{ColorTypes.RGB{FixedPointNumbers.N0f8}}[
    1:
    2:
    3:
    4:
    5:
]
```



Permute the grid by applying the tile flipping rule the desired number of times.

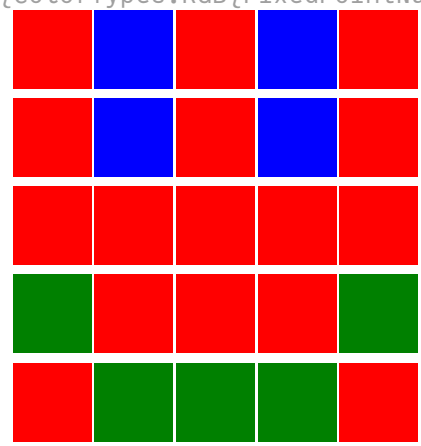
The intermediate results are shown below for a step size of 5.

```
▶[[[0, 2, 0, 2, 0], [0, 2, 0, 2, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 1], [0, 1, 1, 1, 0]], [[0
```

Those same results are shown belown as a grid of colored tiles.
(You may need to expand the output multiple times so that it looks like a grid.)

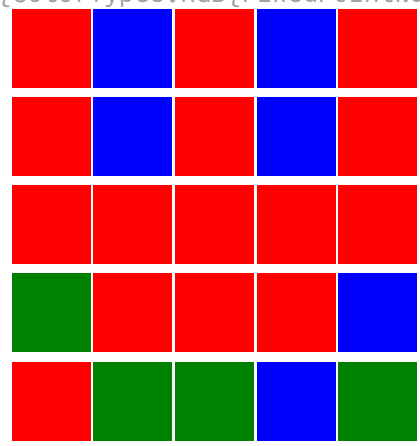
Vector{Vector{ColorTypes.RGB{FixedPointNumbers.N0f8}}}[

1: Vector{ColorTypes.RGB{FixedPointNumbers.N0f8}}[



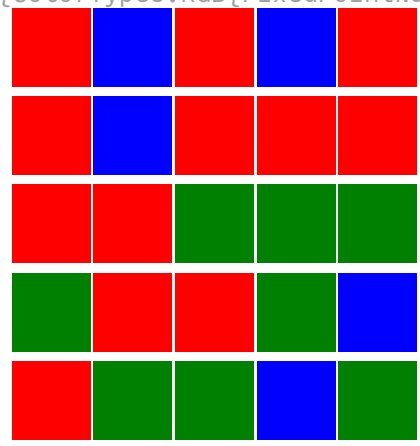
]

2: Vector{ColorTypes.RGB{FixedPointNumbers.N0f8}}[



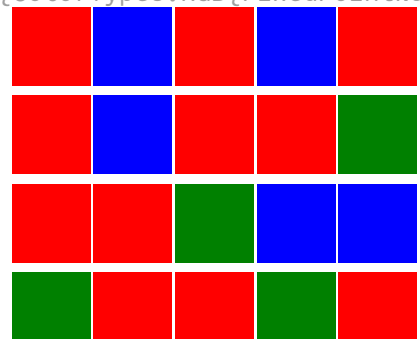
]

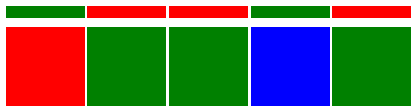
3: Vector{ColorTypes.RGB{FixedPointNumbers.N0f8}}[

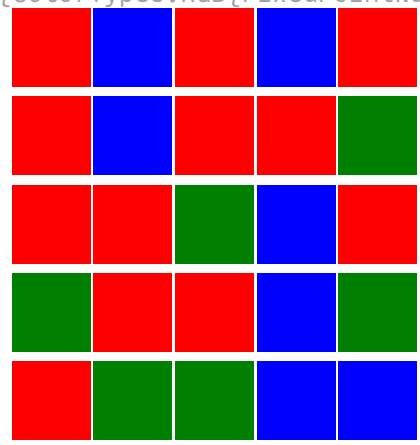


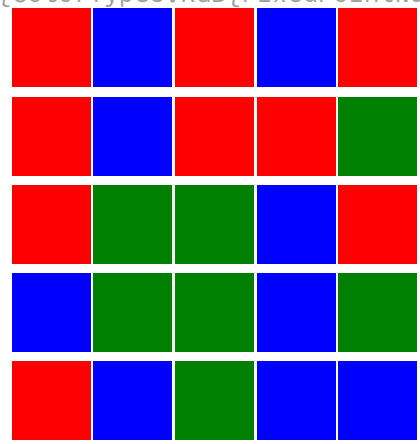
]

4: Vector{ColorTypes.RGB{FixedPointNumbers.N0f8}}[









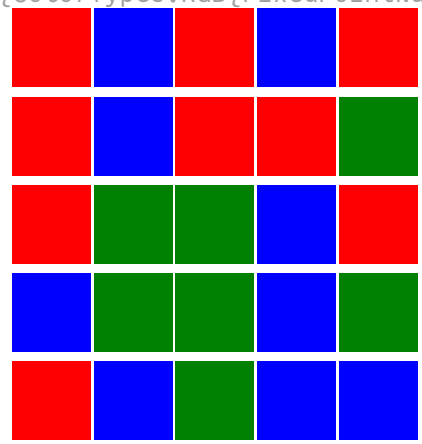
```
]
final =
▶ [[0, 2, 0, 2, 0], [0, 2, 0, 0, 1], [0, 1, 1, 2, 0], [2, 1, 1, 2, 1], [0, 2, 1, 2, 2]]
```

Here we have the final state of the grid.

(You may need to expand the output so that it looks like a grid.)

Vector{ColorTypes.RGB{FixedPointNumbers.N0f8}}[

1:



2:

3:

4:

5:

]

The 25-character string below represents the final scrambled state of the grid.

"02020020010112021121021 22"

Click the button below to scramble the grid until you get a result you like!

Scramble

Appendix

`palette`

The color palette to use when displaying tiles.

```
"The color palette to use when displaying tiles."
palette = [
    colorant"red",
    colorant"green",
    colorant"blue",
]
```

`flipcross`

The relative coordinates for the cross-shaped flipping pattern.

```
"The relative coordinates for the cross-shaped flipping pattern."
flipcross = [(0, -1), (-1, 0), (0, 0), (1, 0), (0, 1)]
```

numbertocolor

```
numbertocolor(number)
```

Convert `number` to its appropriate color value in the default `palette`.

Examples

```
julia> numbertocolor(0)
palette[1]
```

```
"""
    numbertocolor(number)

Convert `number` to its appropriate color value in the default `palette`.

# Examples
```julia-repl
julia> numbertocolor(0)
palette[1]
```
"""
function numbertocolor(number)
    return palette[number + 1]
end
```

`numbertocolor`

```
numbertocolor(number)
```

Convert `number` to its appropriate color value in the default `palette`.

Examples

```
julia> numbertocolor(0)
palette[1]
```

```
numbertocolor(number, palette)
```

Convert `number` to its appropriate color value in `palette`.

Examples

```
julia> numbertocolor(0, [colorant"red", colorant"green", colorant"blue",])
colorant"red"
```

```
"""
    numbertocolor(number, palette)

Convert `number` to its appropriate color value in `palette`.

# Examples
```julia-repl
julia> numbertocolor(0, [colorant"red", colorant"green", colorant"blue",])
colorant"red"
```
"""
function numbertocolor(number, palette)
    return palette[number + 1]
end
```

vectortocolors

```
vectortocolors(vector)
```

Convert `vector` elements to their appropriate color values in the default `palette`.

Examples

```
julia> vectortocolors([0 1 2])
[palette[1] palette[2] palette[3]]
```

```
"""
    vectortocolors(vector)

Convert `vector` elements to their appropriate color values in the default
`palette`.

# Examples
```julia-repl
julia> vectortocolors([0 1 2])
[palette[1] palette[2] palette[3]]
```
"""
function vectortocolors(vector)
    return numbertocolor.(vector)
end
```

`vectortocolors`

```
vectortocolors(vector)
```

Convert `vector` elements to their appropriate color values in the default `palette`.

Examples

```
julia> vectortocolors([0 1 2])
[palette[1] palette[2] palette[3]]
```

```
vectortocolors(number, palette)
```

Convert `vector` elements to their appropriate color values in `palette`.

Examples

```
julia> vectortocolors([0 1 2], [colorant"red", colorant"green", colorant"blu
e",])
[colorant"red" colorant"green" colorant"blue"]
```

```
"""
    vectortocolors(number, palette)

Convert `vector` elements to their appropriate color values in `palette`.

# Examples
```julia-repl
julia> vectortocolors([0 1 2], [colorant"red", colorant"green", colorant"blue",])
[colorant"red" colorant"green" colorant"blue"]
```
"""
function vectortocolors(vector, palette)
    return numbertocolor.(vector, palette)
end
```

shouldbeflipped

```
shouldbeflipped(row, col, targetrow, targetcol)
```

Checks if a tile with coordinates (`row`, `col`) should be flipped based on an interaction with a tile with coordinates (`targetrow`, `targetcol`).

Examples

```
julia> shouldbeflipped(1, 1, 1, 2)
true
```

```
"""
    shouldbeflipped(row, col, targetrow, targetcol)

Checks if a tile with coordinates (`row`, `col`) should be flipped based on an
interaction with a tile with coordinates (`targetrow`, `targetcol`).

# Examples
```julia-repl
julia> shouldbeflipped(1, 1, 1, 2)
true
```
"""
function shouldbeflipped(row, col, targetrow, targetcol)
    (row, col) in map(x -> x .+ (targetrow, targetcol), flipcross)
end
```

permutetile

```
permutetile(matrix, targetrow, targetcol)
```

Flips all relevant tiles in `matrix` based on an interaction with a tile with coordinates (`targetrow`, `targetcol`).

Examples

```
julia> a = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
julia> permutetile(a, 2, 2)
[[0, 1, 0], [1, 1, 1], [0, 1, 0]]
```

```
"""
    permutetile(matrix, targetrow, targetcol)

Flips all relevant tiles in `matrix` based on an interaction with a tile with
coordinates (`targetrow`, `targetcol`).

# Examples
```julia-repl
julia> a = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
julia> permutetile(a, 2, 2)
[[0, 1, 0], [1, 1, 1], [0, 1, 0]]
```
"""
function permutetile(matrix, targetrow, targetcol)
    return map(((col, line),) ->
        map(((row, value),) ->
            shouldbeflipped(row, col, targetrow, targetcol) ? (value + 1) % 3 :
            value, enumerate(line)), enumerate(matrix))
end
```

`createMoves`

```
createMoves(dim, steps, canfliptwice)
```

Creates `steps` number of valid moves for permuting the grid of size `dim`x`dim`. Valid in this case is within the bounds of the grid with duplicate moves limited based on `canfliptwice`.

Examples

```
julia> createMoves(5, 5, false)
[(2, 3), (4, 5), (4, 2), (3, 2), (3, 3)]
```

```
"""
    createMoves(dim, steps, canfliptwice)

Creates `steps` number of valid moves for permuting the grid of size `dim`x`dim`.
Valid in this case is within the bounds of the grid with duplicate moves limited
based on `canfliptwice`.

# Examples
```julia-repl
julia> createMoves(5, 5, false)
[(2, 3), (4, 5), (4, 2), (3, 2), (3, 3)]
```
"""
function createMoves(dim, steps, canfliptwice)
    moves = []
    duplicates = 0

    while (length(moves) - duplicates) < steps
        move = (rand(1:dim), rand(1:dim))
        if count(==(move), moves) < 1
            push!(moves, move)

            if canfliptwice && rand(Bool)
                push!(moves, move)
                duplicates += 1
            end
        end
    end

    return moves
end
```

`createGridStates`

```
createGridStates(matrix, difficulty)
```

Permute `matrix` a number of times based on `difficulty` and keep intermediate states.

Examples

```
julia> a = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
julia> createGridStates(a, "Easy")
[[[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 1, 1], [0, 0, 1], [0, 0, 0]], [[1,
0, 1], [1, 0, 1], [0, 0, 0]], [[1, 0, 1], [0, 0, 1], [1, 1, 0]], [[1, 0, 1],
[0, 0, 0], [1, 0, 1]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]]]
```

```
"""
    creategridstates(matrix, difficulty)

Permute `matrix` a number of times based on `difficulty` and keep intermediate
states.

# Examples
```julia-repl
julia> a = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
julia> creategridstates(a, "Easy")
```
```

```
[[[0, 0, 0], [0, 0, 0], [0, 0, 0]], [[0, 1, 1], [0, 0, 1], [0, 0, 0]], [[1, 0, 1],
[1, 0, 1], [0, 0, 0]], [[1, 0, 1], [0, 0, 1], [1, 1, 0]], [[1, 0, 1], [0, 0, 0],
[1, 0, 1]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]]]
```
"""
function creategridstates(matrix, difficulty)
    dim = size(matrix, 1)
    # Defaults to Easy difficulty rules
    steps = 5
    canfliptwice = false

    if difficulty == "Medium"
        steps = 7
    elseif difficulty == "Hard"
        steps = 9
        canfliptwice = true
    end

    moves = createMoves(dim, steps, canfliptwice)

    states = [matrix]

    for i in 1:length(moves)
        targetrow, targetcol = moves[i]
        push!(states, permutetile(states[i], targetrow, targetcol))
    end

    return states
end
```
```