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

Light Chasing

The purpose of this notebook is to find the moves used for the Light Chasing algorithm in Flippin.

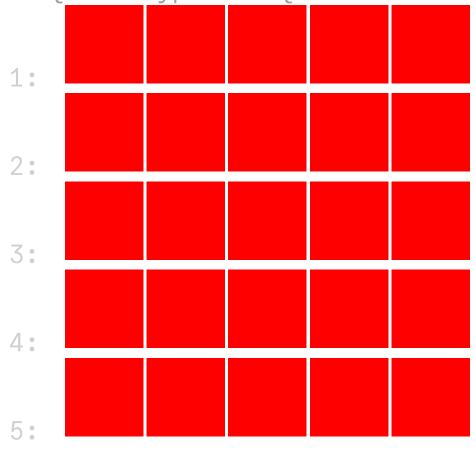
Steps

First, create a base board to use. Any other Flippin board will use the same results so picking one with all red tiles to make following along easier is fine.

```
basevector = ▸[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
base =
▸[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

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



```
]
```

Next, generate the opening gambits for the board. These are all possible tile flips that only use the first row of the board.

```
gambits =
▸[[0, 0, 0, 0, 1], [0, 0, 0, 0, 2], [0, 0, 0, 1, 0], [0, 0, 0, 1, 1], [0, 0, 0, 1, 2], [0, 0
```

```julia
gambits = [parse.(Int, collect(string(i, base=3, pad=5))) for i in 1:242]
```

Now, execute the Light Chasing algorithm and keep track of which gambits map to which final row configurations.

There are 9 unique final row configurations (8 and a no-op). For later simplicity, only track the ones whose gambits require the least amount of tiles flipped.

```
results =
▾Dict{Any, Any}(
    ▸[1, 0, 2, 0, 1] ⇒ ▸[[0, 2, 0, 0, 0], [0, 0, 0, 2, 0], [0, 1, 0, 1, 0]]
    ▸[0, 0, 0, 0, 0] ⇒ ▸[[0, 0, 1, 0, 1], [1, 0, 1, 0, 0]]
    ▸[2, 1, 1, 1, 2] ⇒ ▸[[0, 0, 1, 0, 0]]
    ▸[2, 0, 1, 0, 2] ⇒ ▸[[0, 1, 0, 0, 0], [0, 0, 0, 1, 0]]
    ▸[1, 2, 2, 2, 1] ⇒ ▸[[0, 0, 0, 0, 1], [1, 0, 0, 0, 0]]
    ▸[0, 2, 0, 2, 0] ⇒ ▸[[1, 0, 0, 1, 0], [0, 0, 0, 1, 1], [1, 1, 0, 0, 0], [0, 1, 0, 0,
    ▸[0, 1, 0, 1, 0] ⇒ ▸[[0, 2, 1, 0, 0], [0, 0, 1, 2, 0], [0, 1, 1, 1, 0]]
    ▸[1, 1, 2, 1, 1] ⇒ ▸[[0, 1, 1, 0, 0], [0, 0, 1, 1, 0]]
    ▸[2, 2, 1, 2, 2] ⇒ ▸[[1, 0, 0, 2, 0], [1, 1, 0, 1, 0], [0, 0, 0, 2, 1], [0, 1, 0, 1,
)
```

And that's it! The table below contains the tiles that need to be pressed on the top row to solve for a bottom row configuration at that step in the Light Chasing algorithm. Any of the gambits from the results will work, but I chose the ones I thought would be easiest to remember.

Bottom row offset	Press on top row
1 0 2 0 1	0 1 0 1 0
2 1 1 1 2	0 0 1 0 0
2 0 1 0 2	0 1 0 0 0
1 2 2 2 1	1 0 0 0 0
0 2 0 2 0	1 1 0 0 0
0 1 0 1 0	0 1 1 1 0
1 1 2 1 1	0 1 1 0 0
2 2 1 2 2	1 2 0 0 0

Note: The "Bottom row offset" column in the table means how many places ahead in the color cycle the bottom row of the current board is from the bottom row of the end board.

For the following last row configurations,

End board:

Current board:

🟩🟩🟥🟩🟩

you would use the result associated with $\boxed{2}\boxed{2}\boxed{1}\boxed{2}\boxed{2}$ (i.e. $\boxed{1}\boxed{2}\boxed{0}\boxed{0}\boxed{0}$).

Notebook Functions

```
permuteline (generic function with 1 method)
```

```julia
function permuteline(matrix, moves, lineindex)
    mut = map(copy, matrix)

    for (index, value) in enumerate(moves)
        for _ in 1:value
            mut = permutetile(mut, index, lineindex)
        end
    end

    return mut
end
```

```
generategambitboards (generic function with 1 method)
```

```julia
function generategambitboards(matrix, gambits, lineindex)
    dict = Dict()

    for gambit in 1:length(gambits)
        dict[gambits[gambit]] = permuteline(matrix, gambits[gambit], lineindex)
    end

    return dict
end
```

```
invertline (generic function with 1 method)
```

```julia
function invertline(line)
    return map(x -> (3 - x) % 3, line)
end
```

```
chaselights (generic function with 1 method)
```

```julia
function chaselights(matrix)
    mut = map(copy, matrix)

    for line in 1:4
        inverseline = invertline(mut[line])
        mut = permuteline(mut, inverseline, line + 1)
    end

    return mut
end
```

```
uniquechaselights (generic function with 1 method)
```

```julia
function uniquechaselights(matrix, gambits)
    results = Dict()

    gambits = generategambitboards(base, gambits, 1)

    for (key, value) in gambits
        lastrow = invertline(chaselights(value)[5])
        current = get(results, lastrow, [[3, 3, 3, 3, 3]])
        sumkey = sum(key)
        sumcurrent = sum(current[1])

        if sumkey < sumcurrent
            results[lastrow] = [key]
        elseif sumkey == sumcurrent
            push!(results[lastrow], key)
        end
    end

    return results
end
```

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"]
```

~~~julia
"""
    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-repl
julia> shouldbeflipped(1, 1, 1, 2)
true
```

~~~julia
"""
    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]]
```

~~~julia
"""
    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
```
```