

```
1 begin
2     using LinearAlgebra
3 end
```

Minimal Solution

The purpose of this notebook is to create a utility for finding the minimal solution of a Flippin puzzle (i.e. requires the fewest moves to solve).

This notebook expands on the results of the one describing how to find any solution to a Flippin puzzle if it exists. That notebook is found [here](#).

Prerequisites

Before doing anything, we will introduce the results from the notebook mentioned previously as constants.

A = 25×25 Matrix{Int64}:

```
1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 1 1 1 0 0 0 1 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 0 0 0 1 0 0 0 1 1 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1
```

```
1 A = [
2 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
3 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
4 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
5 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
6 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
7 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0;
8 0 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0;
9 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0;
10 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0;
11 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0;
12 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0;
13 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0;
14 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0;
15 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0;
16 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0;
17 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0;
18 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0;
19 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0;
20 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0 1;
21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1;
22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0;
23 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0;
24 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1;
25 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1;
26 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1;
27 ]
```

p = 3

```
1 p = 3
```

```

▼Matrix{Int64}[
  1: 25×25 Matrix{Int64}:
    1 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 1 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 1 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 1 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 1 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 1 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 1 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 0 0 0 0 0 0 0 0 0 0 0 1 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 1 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 1 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 1 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 1 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 1 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 1

  2: 25×25 Matrix{Int64}:
    1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    1 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    1 2 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    2 1 0 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    2 0 1 2 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    1 1 0 2 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    ⋮
    1 0 1 1 1 2 1 1 0 1 2 2 0 0 1 0 1 0 2 1 0 0 0 0 0
    2 2 2 0 2 2 0 2 2 1 2 0 0 1 1 2 1 0 2 0 1 0 0 0 0
    2 1 2 1 0 0 1 2 0 2 0 2 1 1 1 1 2 0 0 2 0 1 0 0 0
    1 1 2 2 1 1 2 1 1 0 2 1 0 2 1 2 1 2 2 0 1 0 1 0 0
    1 0 0 0 2 2 2 0 1 1 1 2 0 1 2 1 1 0 2 2 0 2 0 1 0
    0 1 0 2 0 2 2 0 1 1 2 1 0 2 1 1 1 0 2 2 2 0 0 0 1

  3: 25×25 Matrix{Int64}:
    1 0 2 1 1 1 0 1 0 1 0 2 0 2 1 2 0 1 1 1 1 1 1 1 0
    0 0 1 2 0 2 0 1 2 0 2 1 0 2 2 1 2 1 2 0 2 1 1 0 1
    0 1 0 0 2 0 0 0 0 0 2 2 2 1 1 0 0 0 2 1 1 1 2 0 0
    0 0 0 1 1 1 2 2 2 2 1 2 0 2 2 2 1 1 0 1 2 0 2 0 2
    0 0 0 0 0 2 1 1 1 1 2 0 1 1 1 1 2 1 2 1 0 1 1 2 0
    0 0 0 0 2 0 0 1 1 2 1 0 0 2 0 0 1 1 0 2 0 1 1 2 2
    0 0 0 0 0 0 0 1 1 2 2 1 1 1 2 0 1 1 1 2 0 2 2 2 2
    ⋮
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 2 2
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1 0 2
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 2 0
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 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 1 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 1
]

```

```
1 D, S, T = finddiagonalmodp(p, A)
```

Steps

We start by defining the start state and end state for the puzzle we are trying to solve.

```
startstate = ▶ [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, 0]
```

```
1 startstate = [  
2     0;  
3     0;  
4     0;  
5     0;  
6     0;  
7     0;  
8     0;  
9     0;  
10    0;  
11    0;  
12    0;  
13    0;  
14    0;  
15    0;  
16    0;  
17    0;  
18    0;  
19    0;  
20    0;  
21    0;  
22    0;  
23    0;  
24    0;  
25    0;  
26    0;  
27 ]
```

```
endstate = ▶ [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

```
1 endstate = [  
2     2;  
3     2;  
4     2;  
5     2;  
6     2;  
7     2;  
8     2;  
9     2;  
10    2;  
11    2;  
12    2;  
13    2;  
14    2;  
15    2;  
16    2;  
17    2;  
18    2;  
19    2;  
20    2;  
21    2;  
22    2;  
23    2;  
24    2;  
25    2;  
26    2;  
27 ]
```

Now we need to consider the null space of \mathbf{A} . All the moves in the null space of \mathbf{A} will result in no net change when applied to a board.

The nullity of \mathbf{A} is 3 so we want to find 3 vectors in the null space of \mathbf{A} that can form the basis for $\text{Null}(\mathbf{A})$. The last 3 rows of the matrix \mathbf{S} found in the other notebook will suffice.

25x25 Matrix{Int64}:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	2	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
⋮				⋮						⋮				⋮				⋮					
1	0	1	1	1	2	1	1	0	1	2	2	0	0	1	0	1	0	2	1	0	0	0	0
2	2	2	0	2	2	0	2	2	1	2	0	0	1	1	2	1	0	2	0	1	0	0	0
2	1	2	1	0	0	1	2	0	2	0	2	1	1	1	2	0	0	2	0	1	0	0	0
1	1	2	2	1	1	2	1	1	0	2	1	0	2	1	2	1	2	2	0	1	0	1	0
1	0	0	0	2	2	2	0	1	1	1	2	0	1	2	1	1	0	2	2	0	2	0	1
0	1	0	2	0	2	2	0	1	1	2	1	0	2	1	1	1	0	2	2	2	0	0	1

1 S

```
v1 = ▶ [1, 1, 2, 2, 1, 1, 2, 1, 1, 0, 2, 1, 0, 2, 1, 2, 1, 2, 2, 0, 1, 0, 1, 0, 0]
```

```
1 v1 = S[23, :]
```

```
v2 = ▶ [1, 0, 0, 0, 2, 2, 2, 0, 1, 1, 1, 2, 0, 1, 2, 1, 1, 0, 2, 2, 0, 2, 0, 1, 0]
```

```
1 v2 = S[24, :]
```

```
v3 = ▶ [0, 1, 0, 2, 0, 2, 2, 0, 1, 1, 2, 1, 0, 2, 1, 1, 1, 0, 2, 2, 2, 0, 0, 0, 1]
```

```
1 v3 = S[25, :]
```

Because $v1$, $v2$, and $v3$ are in the null space of A, adding them (or any linear combination of them) to a solution will produce another valid solution.

Given a solution β , there are 27 total solutions of the form $\beta + c1v1 + c2v2 + c3v3 \pmod{3}$ where $c1$, $c2$, and $c3$ are each either 0, 1, or 2.

Our approach will be to go through the 27 solutions and find the one that uses the fewest moves.

The results below show the number of moves; coefficients for $v1$, $v2$, and $v3$; and minimal solution for a given board.

```
▼Any[
  1: 18
  2: ▶ [1, 0, 2]
  3: ▶ [2, 0, 1, 1, 0, 0, 2, 0, 0, 1, 1, 0, 2, 0, 1, 1, 0, 0, 2, 0, 0, 1, 1, 0, 2]
]
```

```
1 findminimalsolution(startstate, endstate)
```

Notebook Functions

findsolution (generic function with 1 method)

```
1 function findsolution(startstate, endstate)
2     b = mod.(endstate - startstate, p)
3     sb = (S * b) .% p
4     xm = (D * sb) .% p
5     x = (T * xm) .% p
6
7     if (A * x) .% p == b
8         return x
9     else
10         return nothing
11     end
12 end
```

findminimalsolution (generic function with 1 method)

```
1 function findminimalsolution(startstate, endstate)
2     β = findsolution(startstate, endstate)
3     solution = β
4
5     if isnothing(β)
6         return nothing
7     else
8         coefficients = [0, 0, 0]
9         fewestmoves = sum(β)
10
11         for i in 1:26
12             c1, c2, c3 = parse.(Int, collect(string(i, base=3, pad=3)))
13             potentialsolution = (β + (c1 * v1) + (c2 * v2) + (c3 * v3)) .% 3
14             movecount = sum(potentialsolution)
15
16             if movecount < fewestmoves
17                 fewestmoves = movecount
18                 coefficients = [c1, c2, c3]
19                 solution = potentialsolution
20             end
21         end
22
23         return [fewestmoves, coefficients, solution]
24     end
25 end
```

Appendix

swaprows!

```
swaprows!(M::AbstractMatrix, i::Int64, j::Int64)
```

Swap rows *i* and *j* in matrix.

Examples

```
julia> a = [0 0; 0 1]
julia> swaprows!(a, 1, 2)
julia> a
[0 1; 0 0]
```

```
1  """
2      swaprows!(M::AbstractMatrix, i::Int64, j::Int64)
3
4  Swap rows `i` and `j` in `matrix`.
5
6  # Examples
7  ```julia-repl
8  julia> a = [0 0; 0 1]
9  julia> swaprows!(a, 1, 2)
10 julia> a
11 [0 1; 0 0]
12 ```
13 """
14 function swaprows!(M::AbstractMatrix, i::Int64, j::Int64)
15     i == j && return
16     rows = axes(M, 1)
17     @boundscheck i in rows || throw(BoundsError(M, (:,i)))
18     @boundscheck j in rows || throw(BoundsError(M, (:,j)))
19     for k in axes(M, 2)
20         @inbounds M[i, k], M[j, k] = M[j, k], M[i, k]
21     end
22 end
```


swapcols!

```
swapcols!(M::AbstractMatrix, i::Int64, j::Int64)
```

Swap columns *i* and *j* in matrix.

Examples

```
julia> a = [0 0; 0 1]
julia> swapcols!(a, 1, 2)
julia> a
[0 0; 1 0]
```

```
1  """
2      swapcols!(M::AbstractMatrix, i::Int64, j::Int64)
3
4  Swap columns `i` and `j` in `matrix`.
5
6  # Examples
7  ```julia-repl
8  julia> a = [0 0; 0 1]
9  julia> swapcols!(a, 1, 2)
10 julia> a
11 [0 0; 1 0]
12 ```
13 """
14 function swapcols!(M::AbstractMatrix, i::Int64, j::Int64)
15     i == j && return
16     cols = axes(M, 2)
17     @boundscheck i in cols || throw(BoundsError(M, (:,i)))
18     @boundscheck j in cols || throw(BoundsError(M, (:,j)))
19     for k in axes(M, 1)
20         @inbounds M[k, i], M[k, j] = M[k, j], M[k, i]
21     end
22 end
```

scalerowmodp!

```
scalerowmodp!(p::Int64, M::AbstractMatrix, s::Int64, row::Int64)
```

Scale row `row` in matrix by `s` and mod the operation by `p`.

Examples

```
julia> a = [0 0; 0 1]
julia> scalerowmodp!(3, a, 4, 2)
julia> a
[0 0; 0 1]
```

```
1  """
2      scalerowmodp!(p::Int64, M::AbstractMatrix, s::Int64, row::Int64)
3
4  Scale row `row` in `matrix` by `s` and mod the operation by `p`.
5
6  # Examples
7  ```julia-repl
8  julia> a = [0 0; 0 1]
9  julia> scalerowmodp!(3, a, 4, 2)
10 julia> a
11 [0 0; 0 1]
12 ```
13 """
14 function scalerowmodp!(p::Int64, M::AbstractMatrix, s::Int64, row::Int64)
15     sm = mod(s, p)
16     sm == 1 && return
17     rows = axes(M, 1)
18     @boundscheck row in rows || throw(BoundsError(M, (:, row)))
19     for col in axes(M, 2)
20         @inbounds M[row, col] = mod(M[row, col] * sm, p)
21     end
22 end
```

scalecolmodp!

```
scalecolmodp!(p::Int64, M::AbstractMatrix, s::Int64, col::Int64)
```

Scale column `col` in `matrix` by `s` and mod the operation by `p`.

Examples

```
julia> a = [0 0; 0 1]
julia> scalecolmodp!(3, a, 4, 2)
julia> a
[0 0; 0 1]
```

```
1  """
2      scalecolmodp!(p::Int64, M::AbstractMatrix, s::Int64, col::Int64)
3
4  Scale column `col` in `matrix` by `s` and mod the operation by `p`.
5
6  # Examples
7  ```julia-repl
8  julia> a = [0 0; 0 1]
9  julia> scalecolmodp!(3, a, 4, 2)
10 julia> a
11 [0 0; 0 1]
12 ```
13 """
14 function scalecolmodp!(p::Int64, M::AbstractMatrix, s::Int64, col::Int64)
15     sm = mod(s, p)
16     sm == 1 && return
17     cols = axes(M, 2)
18     @boundscheck col in cols || throw(BoundsError(M, (:, col)))
19     for row in axes(M, 1)
20         @inbounds M[row, col] = mod(M[row, col] * sm, p)
21     end
22 end
```

subtractrowmodp!

```
subtractrowmodp!(p::Int64, M::AbstractMatrix, j::Int64, i::Int64, s::Int64=1)
```

Subtract row `i` in matrix from row `j` in matrix after scaling `i` by `s` and mod the operation by `p`.

Examples

```
julia> a = [0 0; 0 1]
julia> subtractrowmodp!(3, a, 1, 2, 2)
julia> a
[0 1; 0 1]
```

```
1  """
2      subtractrowmodp!(p::Int64, M::AbstractMatrix, j::Int64, i::Int64, s::Int64=1)
3
4  Subtract row `i` in `matrix` from row `j` in `matrix` after scaling `i` by `s` and
   mod the operation by `p`.
5
6  # Examples
7  ```julia-repl
8  julia> a = [0 0; 0 1]
9  julia> subtractrowmodp!(3, a, 1, 2, 2)
10 julia> a
11 [0 1; 0 1]
12 ```
13 """
14 function subtractrowmodp!(p::Int64, M::AbstractMatrix, j::Int64, i::Int64,
15 s::Int64=1)
16     sm = mod(s, p)
17     sm == 0 && return
18     rows = axes(M, 1)
19     @boundscheck i in rows || throw(BoundsError(M, (:,i)))
20     @boundscheck j in rows || throw(BoundsError(M, (:,j)))
21     for k in axes(M, 2)
22         @inbounds M[j, k] = mod(M[j, k] - (sm * M[i, k]), p)
23     end
24 end
```

subtractcolmodp!

```
subtractcolmodp!(p::Int64, M::AbstractMatrix, j::Int64, i::Int64, s::Int64=1)
```

Subtract column `i` in matrix from column `j` in matrix after scaling `i` by `s` and mod the operation by `p`.

Examples

```
julia> a = [0 0; 0 1]
julia> subtractcolmodp!(3, a, 1, 2, 2)
julia> a
[0 0; 1 1]
```

```
1  """
2      subtractcolmodp!(p::Int64, M::AbstractMatrix, j::Int64, i::Int64, s::Int64=1)
3
4  Subtract column `i` in `matrix` from column `j` in `matrix` after scaling `i` by
5  `s` and mod the operation by `p`.
6
7  # Examples
8  ```julia-repl
9  julia> a = [0 0; 0 1]
10 julia> subtractcolmodp!(3, a, 1, 2, 2)
11 julia> a
12 [0 0; 1 1]
13 ```
14 """
15 function subtractcolmodp!(p::Int64, M::AbstractMatrix, j::Int64, i::Int64,
16 s::Int64=1)
17     sm = mod(s, p)
18     sm == 0 && return
19     cols = axes(M, 2)
20     @boundscheck i in cols || throw(BoundsError(M, (:,i)))
21     @boundscheck j in cols || throw(BoundsError(M, (:,j)))
22     for k in axes(M, 1)
23         @inbounds M[k, j] = mod(M[k, j] - (sm * M[k, i]), p)
24     end
25 end
```

finddiagonalmodp

```
finddiagonalmodp(p::Int64, M::AbstractMatrix)
```

Given matrix M , find a diagonal matrix D and invertible matrices S and T such that $D \equiv SMT \pmod{p}$. p should be a prime number to ensure appropriate results.

Examples

```
julia> a = [1 1; 1 1]
julia> finddiagonalmodp(3, a)
julia> a
[[1 0; 0 0], [1 0; 2 1], [1 2; 0 1]]
```

```

1  """
2      finddiagonalmodp(p::Int64, M::AbstractMatrix)
3
4  Given matrix `M`, find a diagonal matrix ``D`` and invertible matrices ``S`` and
   ``T`` such that ``D \equiv SMT \pmod{p}``. `p` should be a prime number to ensure
   appropriate results.
5
6  # Examples
7  ```julia-repl
8  julia> a = [1 1; 1 1]
9  julia> finddiagonalmodp(3, a)
10 julia> a
11 [[1 0; 0 0], [1 0; 2 1], [1 2; 0 1]]
12 ```
13 """
14 function finddiagonalmodp(p::Int64, M::AbstractMatrix)
15     A = copy(M) .% p
16     dim = size(A, 1)
17     S = Matrix{Int64}(I, dim, dim)
18     T = Matrix{Int64}(I, dim, dim)
19
20     # Loop for each value on the main diagonal
21     for d in 1:dim
22         pivotcol = 0
23
24         # Find a column that has a value in row d
25         for col in d:dim
26             if A[d, col] != 0

```

```

27         pivotcol = col
28         break
29     end
30 end
31
32 # Skip if no column has a non-zero value in row d
33 if pivotcol == 0
34     continue
35 end
36
37 # Swap pivotcol with column d if different
38 if pivotcol != d
39     swapcols!(A, d, pivotcol)
40     swapcols!(T, d, pivotcol)
41     # Now d is the pivot column
42 end
43
44 # Scale pivot column so that value in row d is 1
45 if A[d, d] != 1
46     s = invmod(A[d, d], p)
47     scalecolmodp!(p, A, s, d)
48     scalecolmodp!(p, T, s, d)
49 end
50
51 # Clear out row value in non-pivot columns
52 for col in (d + 1):dim
53     s = A[d, col]
54
55     # Skip columns where row value is 0
56     if s == 0
57         continue
58     end
59
60     # Subtract the scaled pivot column to clear row value
61     subtractcolmodp!(p, A, col, d, s)
62     subtractcolmodp!(p, T, col, d, s)
63 end
64
65 # Clear out column value in non-pivot rows
66 for row in (d + 1):dim
67     s = A[row, d]
68
69     # Skip rows where column value is 0
70     if s == 0
71         continue
72     end
73
74     # Subtract the scaled pivot row to clear column value
75     subtractrowmodp!(p, A, row, d, s)
76     subtractrowmodp!(p, S, row, d, s)
77 end
78 end
79
80 D = (S * M * T) .% p

```



```
81     return [D, S, T]
82 end
```