```julia
begin
    using LinearAlgebra
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  1  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  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  1  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  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  0  1  0  0  0  1  1
```

```julia
A = [
    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 1 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 1 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 1 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 1 0 0 0 0 0 0 0 0 0 0 0;
    0 0 0 0 1 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 1 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 1 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 1 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 1 0 0 0 0 0 0;
    0 0 0 0 0 0 0 0 0 1 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 1 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 1 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 1 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 1 0;
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 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 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 0 1 0 0 0 1 1;
]
```

```
p = 3
```

```julia
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  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  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  0  0  0  0  0  0
    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  1  2  0  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  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
]
```

```julia
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]
```

```
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;
]
```

```
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]
```

```julia
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;
]
```

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

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

```
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
```

```julia
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]
```

```
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]
```

```
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]
```

```
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]
]
```

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

Notebook Functions

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

```julia
function findsolution(startstate, endstate)
    b = mod.(endstate - startstate, p)
    sb = (S * b) .% p
    xm = (D * sb) .% p
    x = (T * xm) .% p

    if (A * x) .% p == b
        return x
    else
        return nothing
    end
end
```

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

```julia
function findminimalsolution(startstate, endstate)
    β = findsolution(startstate, endstate)
    solution = β

    if isnothing(β)
        return nothing
    else
        coefficients = [0, 0, 0]
        fewestmoves = sum(β)

        for i in 1:26
            c1, c2, c3 = parse.(Int, collect(string(i, base=3, pad=3)))
            potentialsolution = (β + (c1 * v1) + (c2 * v2) + (c3 * v3)) .% 3
            movecount = sum(potentialsolution)

            if movecount < fewestmoves
                fewestmoves = movecount
                coefficients = [c1, c2, c3]
                solution = potentialsolution
            end
        end

        return [fewestmoves, coefficients, solution]
    end
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]
```

~~~
"""
    swaprows!(M::AbstractMatrix, i::Int64, j::Int64)

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

# Examples
```julia-repl
julia> a = [0 0; 0 1]
julia> swaprows!(a, 1, 2)
julia> a
[0 1; 0 0]
```
"""
function swaprows!(M::AbstractMatrix, i::Int64, j::Int64)
    i == j && return
    rows = axes(M, 1)
    @boundscheck i in rows || throw(BoundsError(M, (:,i)))
    @boundscheck j in rows || throw(BoundsError(M, (:,j)))
    for k in axes(M, 2)
        @inbounds M[i, k] ,M[j, k] = M[j, k], M[i, k]
    end
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]
```

~~~
"""
    swapcols!(M::AbstractMatrix, i::Int64, j::Int64)

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

# Examples
```julia-repl
julia> a = [0 0; 0 1]
julia> swapcols!(a, 1, 2)
julia> a
[0 0; 1 0]
```
"""
function swapcols!(M::AbstractMatrix, i::Int64, j::Int64)
    i == j && return
    cols = axes(M, 2)
    @boundscheck i in cols || throw(BoundsError(M, (:,i)))
    @boundscheck j in cols || throw(BoundsError(M, (:,j)))
    for k in axes(M, 1)
        @inbounds M[k, i], M[k, j] = M[k, j], M[k, i]
    end
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]
```

~~~julia
"""
    scalerowmodp!(p::Int64, M::AbstractMatrix, s::Int64, row::Int64)

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

# Examples
```julia-repl
julia> a = [0 0; 0 1]
julia> scalerowmodp!(3, a, 4, 2)
julia> a
[0 0; 0 1]
~~~
"""
function scalerowmodp!(p::Int64, M::AbstractMatrix, s::Int64, row::Int64)
    sm = mod(s, p)
    sm == 1 && return
    rows = axes(M, 1)
    @boundscheck row in rows || throw(BoundsError(M, (:, row)))
    for col in axes(M, 2)
        @inbounds M[row, col] = mod(M[row, col] * sm, p)
    end
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]
```

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

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

# Examples
```julia-repl
julia> a = [0 0; 0 1]
julia> scalecolmodp!(3, a, 4, 2)
julia> a
[0 0; 0 1]
```
"""
function scalecolmodp!(p::Int64, M::AbstractMatrix, s::Int64, col::Int64)
    sm = mod(s, p)
    sm == 1 && return
    cols = axes(M, 2)
    @boundscheck col in cols || throw(BoundsError(M, (:, col)))
    for row in axes(M, 1)
        @inbounds M[row, col] = mod(M[row, col] * sm, p)
    end
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]
```

~~~
"""
    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-repl
julia> a = [0 0; 0 1]
julia> subtractrowmodp!(3, a, 1, 2, 2)
julia> a
[0 1; 0 1]
```
"""
function subtractrowmodp!(p::Int64, M::AbstractMatrix, j::Int64, i::Int64,
s::Int64=1)
    sm = mod(s, p)
    sm == 0 && return
    rows = axes(M, 1)
    @boundscheck i in rows || throw(BoundsError(M, (:,i)))
    @boundscheck j in rows || throw(BoundsError(M, (:,j)))
    for k in axes(M, 2)
        @inbounds M[j, k] = mod(M[j, k] - (sm * M[i, k]), p)
    end
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]
```

~~~julia
"""
    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-repl
julia> a = [0 0; 0 1]
julia> subtractcolmodp!(3, a, 1, 2, 2)
julia> a
[0 0; 1 1]
~~~
"""
function subtractcolmodp!(p::Int64, M::AbstractMatrix, j::Int64, i::Int64,
s::Int64=1)
    sm = mod(s, p)
    sm == 0 && return
    cols = axes(M, 2)
    @boundscheck i in cols || throw(BoundsError(M, (:,i)))
    @boundscheck j in cols || throw(BoundsError(M, (:,j)))
    for k in axes(M, 1)
        @inbounds M[k, j] = mod(M[k, j] - (sm * M[k, i]), p)
    end
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]]
```

~~~julia
"""
    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-repl
julia> a = [1 1; 1 1]
julia> finddiagonalmodp(3, a)
julia> a
[[1 0; 0 0], [1 0; 2 1], [1 2; 0 1]]
~~~
"""
function finddiagonalmodp(p::Int64, M::AbstractMatrix)
    A = copy(M) .% p
    dim = size(A, 1)
    S = Matrix{Int64}(I, dim, dim)
    T = Matrix{Int64}(I, dim, dim)

    # Loop for each value on the main diagonal
    for d in 1:dim
        pivotcol = 0

        # Find a column that has a value in row d
        for col in d:dim
            if A[d, col] != 0
```
```

```julia
                pivotcol = col
                break
            end
        end

        # Skip if no column has a non-zero value in row d
        if pivotcol == 0
            continue
        end

        # Swap pivotcol with column d if different
        if pivotcol != d
            swapcols!(A, d, pivotcol)
            swapcols!(T, d, pivotcol)
            # Now d is the pivot column
        end

        # Scale pivot column so that value in row d is 1
        if A[d, d] != 1
            s = invmod(A[d, d], p)
            scalecolmodp!(p, A, s, d)
            scalecolmodp!(p, T, s, d)
        end

        # Clear out row value in non-pivot columns
        for col in (d + 1):dim
            s = A[d, col]

            # Skip columns where row value is 0
            if s == 0
                continue
            end

            # Subtract the scaled pivot column to clear row value
            subtractcolmodp!(p, A, col, d, s)
            subtractcolmodp!(p, T, col, d, s)
        end

        # Clear out column value in non-pivot rows
        for row in (d + 1):dim
            s = A[row, d]

            # Skip rows where column value is 0
            if s == 0
                continue
            end

            # Subtract the scaled pivot row to clear column value
            subtractrowmodp!(p, A, row, d, s)
            subtractrowmodp!(p, S, row, d, s)
        end
    end

    D = (S * M * T) .% p
    return [D, S, T]
end
```