```
begin
    using LinearAlgebra
end
```

Tile Matrix Util

The purpose of this notebook is to create a utility for finding the solvability of a Flippin puzzle.

The steps used largely follow the approach detailed here for solving a system of linear equations mod n.

Steps

We start by defining a 25×25 matrix representing the ruleset of Flippin. Values of this matrix are elements of $\mathbb{Z}_3$. In other words, they can hold the values 0, 1, or 2 as there are 3 possible color states for tiles and values wrap back around.

Each row of the matrix holds coefficients for an equation defining the current value of a tile on the board. The variables in those equations represent the number of actions taken on the corresponding tile. The first row of the matrix represents the value of the leftmost tile of the first row of the board. The sixth row of the matrix represents the value of the leftmost tile of the second row of the board. Columns of the matrix similarly enumerate tiles in the board moving left-to-right and row-by-row.

Taking the first row of the matrix as an example, the values in columns 1, 2, and 6 mean that the value of this upper leftmost tile of the board can be impacted by itself and the tiles directly to the right of it and below it. The coefficients are all 1 because every action on one of those tiles advances the state of this tile by 1.

```
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
```

```
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 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 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;
]
```

Here we define the modulus for the calculations. The fact that this value is prime makes life a lot easier because it means that every value we will see in the matrix will have a corresponding (and easy to find) multiplicative inverse in $\mathbb{Z}_3$.

```
p = 3
```

```
p = 3
```

Given our ruleset matrix A, the approach here is to find a diagonal matrix D and invertible matrices S and T such that $D = SAT$. This will then allow us to confirm if a given starting board state can

reach a given end board state.

```
▸ [25×25 Matrix{Int64}:                                                , 25×25 Matr
    1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  (
```

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

Do a quick sanity check here.

```
true
```

```
1 (S * A * T) .% p == D
```

Now that we have the matrices, let's use them to solve our problem!

The linear system we are trying to solve looks like $Ax \equiv b \pmod 3$ where A is the ruleset matrix, x is the column vector that "*solves*" the system, and b is the column vector representing the desired end state of the board. Not every end state has a solution so we will have to plug any x values we find back into the equation to confirm that they work. There may also exist other solutions that solve the puzzle in fewer steps.

It is important to note that we cannot just plug any end state into the equation unless the start state is a column vector full of zeros. This is because the solution method assumes that we are starting from such a start state. In order to right this issue, we can take the difference of our desired end state and desired start state (mod 3 of course) and then use **that** as the value of b.

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

```
startstate = [
    0;
    2;
    2;
    1;
    0;
    2;
    2;
    1;
    1;
    1;
    1;
    2;
    0;
    0;
    1;
    0;
    1;
    1;
    1;
    2;
    2;
    0;
    1;
    0;
    0;
]
```

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

```
endstate = [
    2;
    1;
    2;
    1;
    2;
    0;
    0;
    0;
    0;
    0;
    0;
    0;
    0;
    0;
    2;
    0;
    0;
    0;
    2;
    2;
    2;
    0;
    2;
    2;
]
```

Below we find the value of $\boldsymbol{x}$ using a series of substitions and matrix multiplications.

The blog post mentioned above does a good job of explaining the solution logic and I would highly recommend giving it a read.

Of note, I ensure that the values on the main diagonal of $\boldsymbol{D}$ are at most $\mathbf{1}$ to allow me to just mutilpy $\boldsymbol{Sb}$ by $\boldsymbol{D}$ to find `xm` ($\boldsymbol{x'}$ in the blog post).

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

```
b = mod.(endstate - startstate, p)
```

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

```
sb = (S * b) .% p
```

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

```
xm = (D * sb) .% p
```

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

```
1  x = (T * xm) .% p
```

Confirm that the found solution works.

```
true
```

```
1  (A * x) .% p == b
```

Now that we have a method for finding solutions, I wanted to be able to apply it easily in JavaScript.

In Julia, the column vector and matrix operations are fine (because the language is designed around doing mathy stuff), but basic JavaScript is not so well-equipped. The **basic** part is important because I do not want to download a bulky library for essentially a simple one-off action.

Using row vectors (essentially JavaScript arrays) seemed like it would be the easiest way to implement this solution but did require some adjustments to the matrices. Namely, I transposed the matrices and rearranged the equations to use left multiplication.

```
true
```

```
1  (T * D * S * b) .% p == x
```

```
true
```

```
1  (b' * S' * D' * T') .% p == x'
```

```
true
```

```
1  (x' * A') .% p == b'
```

Below are the transposed matrices I copied into my JavaScript code.

```
1  println(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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 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 println(D')
```

```
[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 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 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 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 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 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 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 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 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 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 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]
```

```
1 println(S')
```

```
[1 2 1 1 2 2 1 2 0 2 1 0 0 2 1 0 1 1 0 1 2 2 1 1 0; 0 1 2 2 1 0 1 2 2 1 0 2
0 2 2 0 2 1 2 0 2 1 1 0 1; 0 0 1 0 0 1 0 0 2 2 0 0 1 1 0 2 2 0 0 1 2 2 2 0 0;
0 0 0 1 2 2 2 0 1 2 1 0 0 2 1 2 0 1 1 1 0 1 2 0 2; 0 0 0 0 1 0 1 0 2 0 0 2 2 1
2 1 2 1 2 1 2 0 1 2 0; 0 0 0 0 0 1 1 2 1 0 2 1 0 2 0 0 0 1 1 2 2 0 1 2 2; 0 0
0 0 0 0 1 2 2 1 2 1 2 1 0 1 1 1 1 1 0 1 2 2 2; 0 0 0 0 0 0 0 1 1 1 2 1 2 1 0 2
2 1 0 1 2 2 1 0 0; 0 0 0 0 0 0 0 0 1 2 2 1 0 2 0 1 2 1 0 0 2 0 1 1 1; 0 0 0 0
0 0 0 0 0 1 2 1 1 0 0 0 1 1 0 1 1 2 0 1 1; 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 1 0
1 2 2 0 2 1 2; 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 1 0 1 2 2 2 0 2 1 2 1; 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 2 0 2 0 0 1 0 0 0; 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 1 1 2 2 0
1 1 2 1 2; 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 2 1; 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 2 0 2 1 2 1 1; 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 2
1 1 1; 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 2 0 0; 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 2 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 0 2 0 2
2; 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 2; 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 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]
```

```
println(T')
```

```
[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 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 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; 1 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 0; 1 2 0 1 2 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 1 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; 0 2 0 2 1 1 1 2 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0; 1 0 0 2
1 2 2 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0; 0 2 2 1 2 1 2 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0; 2 1 2 2 0 0 1 1 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0; 0 0 2 0 1 0
1 1 0 2 2 1 2 0 0 0 0 0 0 0 0 0 0 0 0; 2 2 1 2 1 2 1 1 2 0 1 0 1 1 0 0 0 0 0 0
0 0 0 0 0; 1 2 1 2 1 0 2 1 2 0 0 1 0 1 0 2 0 0 0 0 0 0 0 0 0; 2 1 0 2 1 0 0 0
0 0 1 2 0 1 2 0 0 0 0 0 0 0 0 0 0; 0 2 0 1 2 1 1 0 0 0 1 2 2 2 1 2 0 0 1 0 0 0
0 0 0; 1 1 0 1 1 1 1 1 1 1 0 2 0 2 0 0 0 1 0 0 0 0 0 0 0; 1 2 2 0 2 0 1 2 2 1
1 1 2 1 1 1 1 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; 1 2 1 2 0 0 2 1 0 1 0 1 2 2 2 2 1 0 0 1 0 2 0 0 0; 1 1 1 0 1 1 0 1 1 2 1 0
0 2 2 1 2 0 1 0 2 0 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]
```

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]
```

```
"""
    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]
```

```
"""
    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]]
```

```
"""
    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
                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]
```

```
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
```