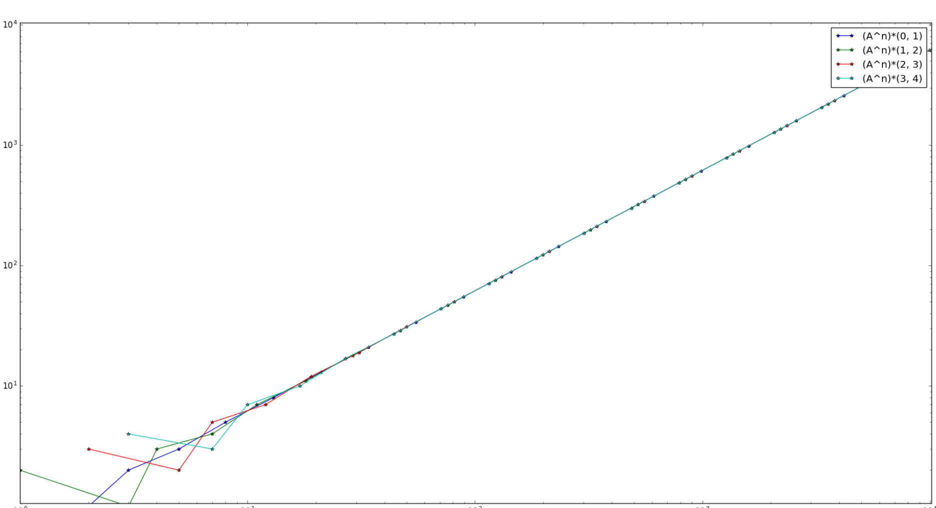
```
No need for fancy tools here...

1)

a - this is an overdetermined system, with more rows than columns, but the vecto
r [0,0,0,0] is a homogenous solution

b - this is an underdetermined system, with more variables than equations

c - overdetermined system, but with no solutions, as 0*x+0*y+0*z=4 is unsatisfya
ble

d - underdetermined system, with no solutions - more equations than variables
```

```
Negligible fancyness here, too...

2

a) eigen_vs(Arr('1 2; 2 1')) -> (-3,1)

b) eigen_vs(Arr('1 0 1; 0 1 2; 0 1 -1')) -> (1, sqrt(3), sqrt(-3))

c) eigenvalues are merely your diagonal - if you frame it as (D-l*I)*v = 0...

 -- it's pretty clear to see that your eigenvalues (l, because typing lambda is
hard)
        are just organised on your diagonal, the values that are zeroed
        when subtracted with your eigenvalues on a diagonal

d) eigen_vs(Arr('1 2 0 0 0; 2 1 0 0 0; 0 0 1 0 1; 0 0 0 1 2; 0 0 1 2 -1'))

  -> (3, -1, 1, sqrt(-6), sqrt(6))

e) aha, this is one of those symmetry tricks used by computers to Make Go Fast

 - the eigenvalues of a block diagonal matrix look like the sum of the set of th
e two eigenvalues, plus or minus a scale factor or so

- if we frame eigenvalues as the values subtracted from a fully diagonalised mat
rix to render it uninvertable, the general construct referred to above, then any
 values that render one portion of a diagonalised matrix uninvertable will rende
r the whole matrix uninvertable

this is a 'thought' experiment illustration...
```

```
A)

A = Arr('5/8 1/8 1/8; 1/3 0 1/3; 1/24 21/24 13/24')

There's not really much to be said here.

Our dominant eigenvalue is 1, associated with the eigenvector [1, 1, 2],
    in which half the blame is assigned to the executive branch

B) well, there's no nice way to put this, besides.... 1/2(sqrt(5)+1)

the ratio of branches to twigs is 1:phi as t->oo
```



(A^n)*(0, 1)
(A^n)*(1, 2)
(A^n)*(2, 3)
(A^n)*(3, 4)

```
from lin1 import *
import itertools
A = Arr('1 1; 1 0')

def apply_matrix(initial_conditions, matrix):
    val = initial_conditions
    while True:
        yield val
        val = mul(matrix, val)

def get_first_N(N, initial_conditions, matrix):
    return [x[1] for x in itertools.takewhile(lambda x: x[0] < N,
        enumerate(apply_matrix(initial_conditions, matrix)))]


import matplotlib
from matplotlib import pyplot
for x0 in zip(range(0,6),range(1,5)):
    xs = get_first_N(50, x0, A)
    plot_A = pyplot.loglog([x[0] for x in xs], [x[1] for x in xs], '-*', label='
(A^n)*'+str(x0))
pyplot.legend()
pyplot.show()
```

As past-me helpfully reduced this process....

```
np.dot(np.dot(eigenvecs_as_columns, eigenvals_on_eigenbasis), np.linalg.inv(eige
nvecs_as_columns))
```

this is pretty intuitive now, although it wasn't then -

we use our eigenvectors to create a linear combination of our variable on orthog onal axes, apply our transformation matrix an arbitrary number of times to advan ce the process, and apply the inverse of our change-of-basis eigenvectors-as-col umns matrix to reset

P is merely our eigenvectors laminated as columns:

```
>>> Arr('1 1 0; 1 0 1; 2 -1 -1')
array([[ 1,  1,  0],
       [ 1,  0,  1],
       [ 2, -1, -1]])
```

D is our eigenvalues dotted with our identity matrix, or something such that it' s a diagonal matrix comprised of our eigenvalues

```
>>> Arr('1 0 0; 0 1/2 0; 0 0 -1/3')
array([[ 1.        ,  0.        ,  0.        ],
       [ 0.        ,  0.5       ,  0.        ],
       [ 0.        ,  0.        , -0.33333333]])
```

to find P**-1:

```
>>> inverse(Arr('1 1 0; 1 0 1; 2 -1 -1'))
array([[ 0.25,  0.25,  0.25],
       [ 0.75, -0.25, -0.25],
       [-0.25,  0.75, -0.25]])
```