HW 3 - STAT 201A

Bobby Thompson

1.

a. The matrix P for this Markov process would look like:

$$P = \begin{pmatrix} .2 & .7 & .1 \\ .2 & .5 & .3 \\ .2 & .4 & .4 \end{pmatrix}$$

b.We will simulate 10 steps through our chain, starting from $X_0 = 1$.

```
import numpy as np
np.random.seed(1)

P = np.array([[.2,.7,.1],[.2,.5,.3],[.2,.4,.4]])
x_0 = 1
state_options = P[x_0-1]

for _ in range(0,10):
    x_next = np.random.choice([1,2,3], p=state_options)
    print(x_next)
    state_options = P[x_next-1]
```

```
2
3
1
2
1
1
1
2
2
2
```

2.

a. What we want to recognize here is that since we are solving for $(P^T - I)\pi_\infty = 0$, we are solving for the nullspace of the matrix given by $P^T - I$. One thing to note is that, when find the stationary solution for Markov processes, we have to normalize the output or it will not be correct. This should be fairly straight forward to do numerically.

```
from scipy.linalg import eig, null_space

M = P.T - np.eye(3)
sol = null_space(M)
normalized_sol = sol/np.sum(sol)
normalized_sol
```

```
array([[0.2       ],
       [0.51111111],
       [0.28888889]])
```

And we can even test that this is the stationary distribution by seeing what happens when we apply it to Markov process P.

```
normalized_sol.T@P
```

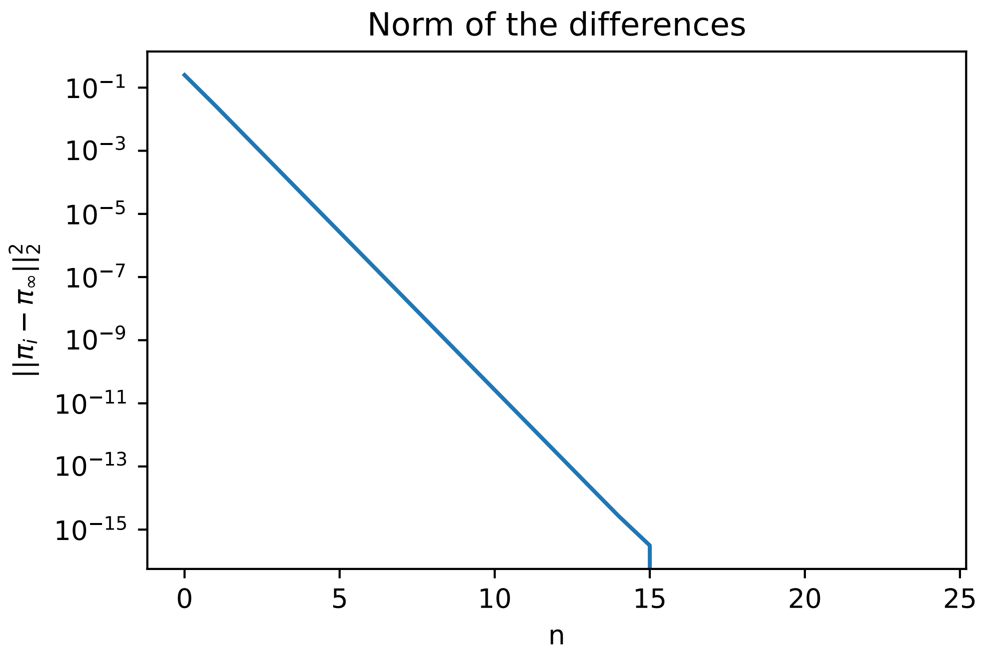
```
array([[0.2       , 0.51111111, 0.28888889]])
```

b. I wrote a function to find the norm diffs through each iteration, and we will use this to plot our distributions.

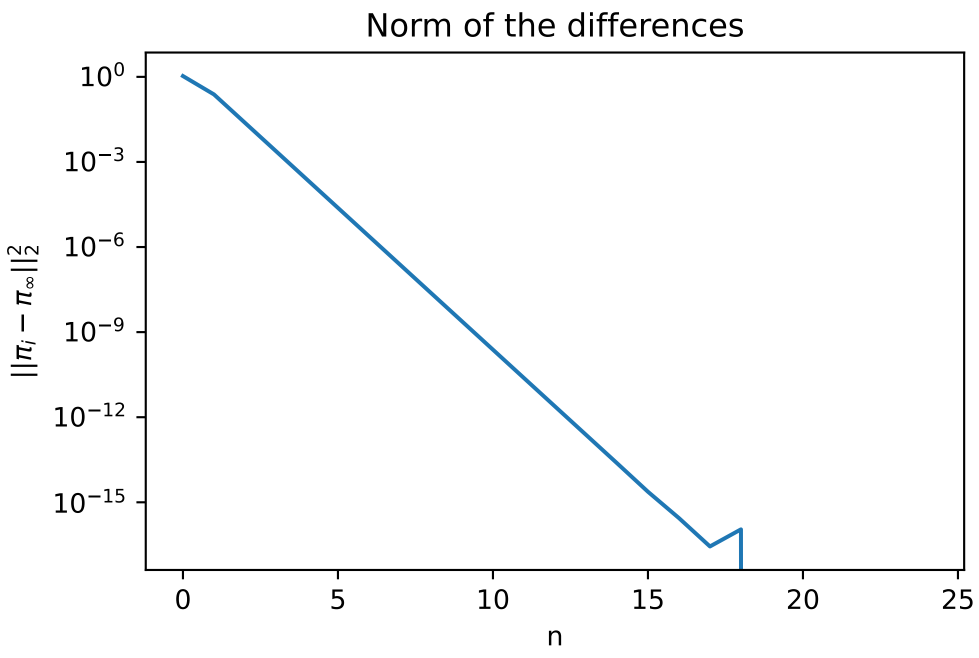
```
from scipy.linalg import norm

i_ = np.arange(0,25,1)
def calculate_norm_diffs(initial_state):
    pi_0 = initial_state
    norm_diff = []
    for i in i_:
        pi_i = pi_0@P
        pi_norm = norm(pi_i - pi_0)
        norm_diff.append(pi_norm)
        pi_0 = pi_i
    return norm_diff
```

So let's try it with a couple of starting values

```python
import matplotlib.pyplot as plt
init = [.33,.33,.34]

norm_diffs = calculate_norm_diffs(init)

plt.plot(i_, norm_diffs)
plt.yscale('log')
plt.title('Norm of the differences')
plt.xlabel('n')
plt.ylabel(r'$|| \pi_{i} - \pi_{\infty}||_2^2$');
```



```python
init = [1,0,0]

norm_diffs = calculate_norm_diffs(init)

plt.plot(i_, norm_diffs)
plt.yscale('log')
plt.title('Norm of the differences')
plt.xlabel('n')
plt.ylabel(r'$|| \pi_{i} - \pi_{\infty}||_2^2$');
```



So it looks like they all eventually converge and our graphs are what we want, but can we go one step further? I want to know if they always converge (as long as the sum of the starting state is 1) and if they specifically converge to the staitionary distribution. Let's try and find out

```
#making a function that creates a random 3 part
#distribution to show that they all converge eventually
def choose_three_numbers_sum_to_one():
    first_select = np.random.uniform(0, 1.0)
    curr_total = 1-first_select
    second_select = np.random.uniform(0,curr_total)
    third_select = curr_total - second_select
    return [first_select,second_select,third_select]

n=10000
convergence_counter = 0
#placeholder, at the end this should be the stationary solution
final_dist = [0,0,0]

for i in range(0,n):
    pi_0 = choose_three_numbers_sum_to_one()
    norm_diff = []
    while True:
        pi_i = pi_0@P
        pi_norm = norm(pi_i - pi_0)
        if pi_norm == 0.0:
            if pi_i.all() == normalized_sol.all():
              convergence_counter += 1
            break
        pi_0 = pi_i
    final_dist = pi_0
print(final_dist)
print(f'how often any state converged to the stationary distribution: {(convergence_counte
```

```
[0.2        0.51111111 0.28888889]
how often any state converged to the stationary distribution: 100.0%
```

So given any starting state, we ended up with the stationary distribution. Very cool!

3.

a. Pretty straight forward. Let's make a function to execuate our Markov simulation.

```
P = np.array([[.2,.7,.1],[.2,.5,.3],[.2,.4,.4]])

def markov_sim(starting,n):
    results = []
    for _ in range(n):
        steps_t = 0
        x_curr = starting
        while True:
            #based on 0 indexing we decrement x_curr
            state_options = P[x_curr-1]
            x_curr = np.random.choice([1,2,3], p=state_options)
            steps_t += 1
            if x_curr == 3:
                results.append(steps_t)
                break
    return results
```

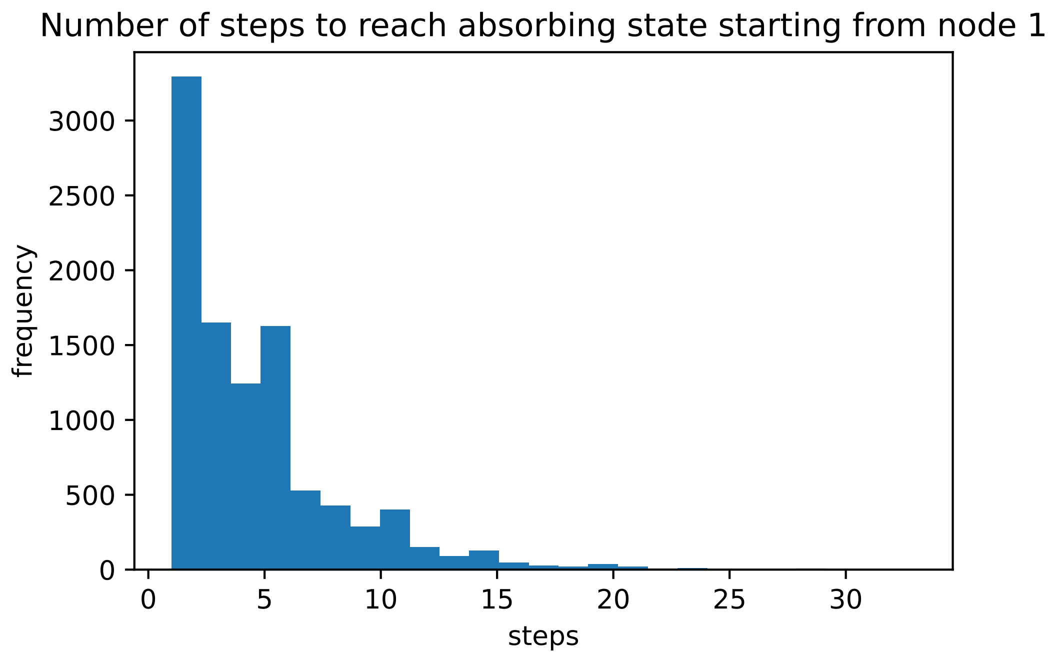
And we will use this to start from both nodes 1 and 2, 10k sims each.

```
start_from_one = markov_sim(1,10000)
start_from_two = markov_sim(2,10000)
```

Let's get our information for starting from 1.

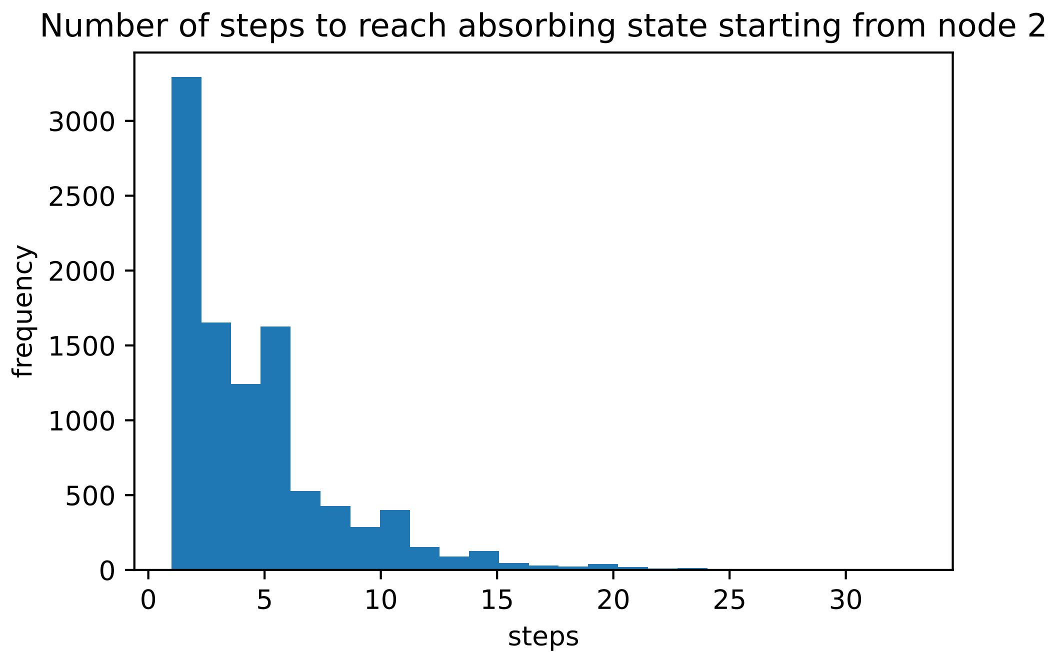
```
print(f'Mean number of steps to reach node 3 starting from 1: {np.mean(start_from_one)}')
plt.hist(start_from_one, bins=25)
plt.title('Number of steps to reach absorbing state starting from node 1')
plt.xlabel('steps')
plt.ylabel('frequency');
```

```
Mean number of steps to reach node 3 starting from 1: 4.6474
```



```
print(f'Mean number of steps to reach node 3 starting from 2: {np.mean(start_from_two)}')
plt.hist(start_from_one, bins=25)
plt.title('Number of steps to reach absorbing state starting from node 2')
plt.xlabel('steps')
plt.ylabel('frequency');
```

```
Mean number of steps to reach node 3 starting from 2: 3.8552
```



b. Now let's solve part b. I will solve this analytically, since I thought it wasn't so bad. We know:

$\mu_i = 1 + \sum_{j=1}^{3} p_{ij}\mu_j$

$\mu_i = E(T_i)$

$T_3 = 0$

So

$\mu_1 = 1 + \sum_{j=1}^{3} p_{1j}\mu_j = 1 + p_{11}\mu_1 + p_{12}\mu_2 + p_{13}\mu_3$

$\mu_2 = 1 + \sum_{j=1}^{3} p_{2j}\mu_j = 1 + p_{21}\mu_1 + p_{22}\mu_2 + p_{23}\mu_3$

$\mu_3 = E(T_3) = E(0) = 0$

So we can eliminate the μ_3 terms, leaving us with

$1 = \mu_1 - p_{11}\mu_1 - p_{12}\mu_2$

$1 = \mu_2 - p_{21}\mu_1 - p_{22}\mu_2$

And when we plug in our values from our matrix P

$1 = .8\mu_1 - .7\mu_2$

$1 = -.2\mu_1 - .5\mu_2$

and this system is realtively simple to solve. When we do we get

$\mu_1 = 4.61538$

$\mu_2 = 3.84615$

Which is very close to our numerical solution from earlier. I see this as an absolute win.