In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
# add here the libraries you need
```

In [2]:

```
import my_methods
```

Discrete Time Markov Chains - part 2

This is an exercise notebook on DTMCs.

Remember to revise of the lecture on DTMC before attempting to solve it!

1. Simulation of DTMC

Write a method that simulates a DTMC for `n` steps, where `n` is a parameter of the method, and returns the whole trajectory as output.

In [ ]:

```

```

In [3]:

```
def simulate_trajectory(transition_matrix,initial_prob,T):

    transition_matrix = np.asarray(transition_matrix)
    traj = np.zeros(T+1, dtype = int) # da ricordare

    t = 0
    traj[0] = my_methods.sample_from_prob_vector(initial_prob)

    while t < T:
        traj[t+1]= my_methods.sample_from_prob_vector(transition_matrix[traj[t
]])
        t += 1

    return traj
```

In [4]:

```
transition_die = np.zeros((13,13))

transition_die[7:13,7:13] = np.eye(6)

transition_die[0,1:3] = 0.5
transition_die[1,3:5] = 0.5
transition_die[2,5:7] = 0.5
transition_die[3,1] = 0.5
transition_die[3,7] = 0.5
transition_die[4,8:10] = 0.5
transition_die[5,10:12] = 0.5
transition_die[6,12] = 0.5
transition_die[6,2] = 0.5

init_die = np.zeros(13)
init_die[0] = 1

dice_dict = {"transition_matrix": transition_die, "initial_prob": init_die}
```

In [5]:

```
import pickle
filename = 'dice_model.pickle'

writefile = open(filename,'wb')
pickle.dump(dice_dict, writefile)
writefile.close()
```

In [6]:

```
readfile = open(filename,'rb')
imported_dict = pickle.load(readfile)
readfile.close()
```

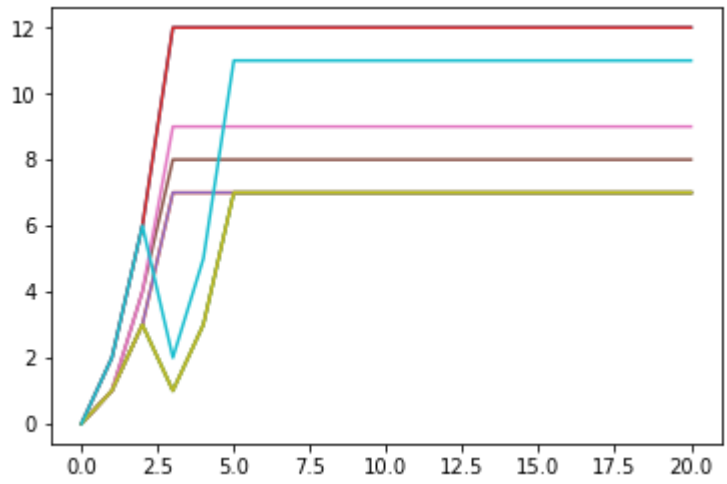
In [ ]:

```

```

In [17]:

```
for t in range(10):
    die_traj = simulate_trajectory(transition_die, init_die, 20)
    plt.plot(np.arange(0,21),die_traj)
```



2. Statistical analysis

Write methods for:

- 2.1. computing the average of a function `f` of the state space, at time step `n`.
- 2.2. computing the probability of reaching a target region `A` of the state space by time step `n`. Both methods should use simulation, and return an estimate and a confidence interval at a specified confidence level `alpha` (0.95% by default).

In [18]:

```
def eval_func_n(f, transition_matrix,initial_prob, n):

    traj = simulate_trajectory(transition_matrix,initial_prob,n)

    return f(traj[-1])
```

In [19]:

```
func = lambda z: z+1

sample_size = 1000
Y = np.zeros(sample_size)
for i in range(sample_size):
    Y[i] = eval_func_n(func, transition_die, init_die, 10)

print(np.mean(Y))
```

```
10.45
```

In [20]:

```
from scipy.stats import norm
alpha = 0.05
Zalpha = norm.ppf(1-(alpha)/2) #Percent Point Function (inverse of cdf — percent
iles)
```

Note: definition of confidence interval is $\overline{Y}_N \pm \frac{1}{\sqrt{N}} \dot{Z}_\alpha \sqrt{\overline{S_N^2}}$.

In [21]:

```
def eval_func_over_traj(g, transition_matrix,initial_prob, m):

    traj = simulate_trajectory(transition_matrix,initial_prob,m)

    return g(traj)
```

In [22]:

```
def reach(traj, A):
    for t in traj:
        if t in A:
            return 1
    return 0
```

In [23]:

```
A = [0]

R = lambda traj: reach(traj,A)

N = 100
Q = np.zeros(N)
for i in range(N):
    Q[i] = eval_func_over_traj(R, transition_die,init_die, 10)

Qm = np.mean(Q)
Qs = np.sum((Q-Qm)**2)/(N-1)

Ic = [Qm-Zalpha*np.sqrt(Qs)/np.sqrt(N), Qm, Qm+Zalpha*np.sqrt(Qs)/np.sqrt(N)]

print(Ic)
```

```
[1.0, 1.0, 1.0]
```

3. Branching chain

Consider a population, in which each individual at each generation independently gives birth to k individuals with probability p_k. These will be the members of the next generation. Assume $k \in \{-1, 0, 1, 2\}$.

Optional: under which conditions the population will become extinct?

In [24]:

```
def sample_from_prob_vector(p):
    n = p.shape[0]
    S = np.cumsum(p)
    U = np.random.uniform(0,1) #usare il punto uno

    for i in range(n):
        if S[i] > U:
            sampled_index = i
            break

    return int(sampled_index)
```

In [25]:

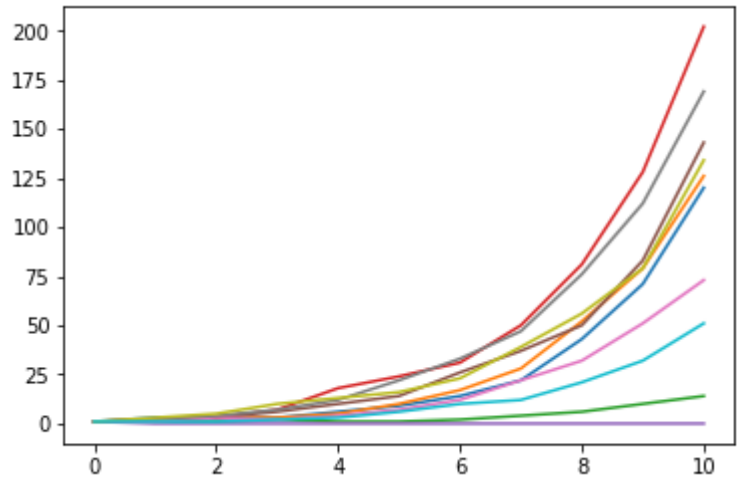
```
def branching(p, T):
    popul = np.ones(T+1, dtype=int)
    for t in range(T):
        tmp_counter = 0
        for indiv in range(popul[t]):
            tmp_counter += sample_from_prob_vector(p)-1
        popul[t+1] = popul[t]+tmp_counter

    return popul
```

In [26]:

```
p = np.array([0.1, 0.4, 0.3, 0.2])
T = 10

for i in range(10):
    popul = branching(p, T)
    plt.plot(np.linspace(0,T,T+1),popul)
plt.show()
```



In [ ]:

```

```

In [ ]:

```

```