Simulation

Assignment 3.2 – Importance sampling

Ari Viitala 432568

In [55]:
```
import numpy as np
```

Classic Monte-Carlo

In [122]:
```
def path(c):
    p = 1.0 / (1 + 5/4.0)
    #the length of the que
    state = 1
    while state > 0 and state < c + 1:
        if np.random.random() < p:
            #if the que length increases we add a person
            state += 1
        else:
            #else we substract a person
            state -= 1
    #if we hit 0 we return zero, else we return 1
    if state == 0:
        return 0
    else:
        return 1
```

In [121]:
```
#different target values
c3 = 3
c5 = 5
c7 = 7
#counters for different target values
counter3 = 0
counter5 = 0
counter7 = 0
iterations = 100000
for i in range(0, iterations):
    #for each iteration calculate one instace of all the different targets
    counter3 += path(c3)
    counter5 += path(c5)
    counter7 += path(c7)

#print the estimared probabilities
print("Probability of queue hitting 3: " + str(counter3 / iterations))
print("Probability of queue hitting 5: " + str(counter5 / iterations))
print("Probability of queue hitting 7: " + str(counter7 / iterations))
```

```
Probability of queue hitting 3: 0.17448
Probability of queue hitting 5: 0.08813
Probability of queue hitting 7: 0.05078
```

The algortihm seems to be OK, since the c = 3 case gives the correct result.

Importance sampling

In [74]:
```
def IS(c):
    p_star = (5.0 / 4.0) / (1 + 5/4.0)
    p = 1.0 / (1 + 5/4.0)
    state = 1
    arrivals = 0
    completions = 0
    #first simulate the queue and save the arrivals and completions
    while state > 0 and state < c + 1:
        if np.random.random() < p_star:
            state += 1
            arrivals += 1
        else:
            state -= 1
            completions += 1

    #returns zero or the term inside the summation in the importance sampling formula
    if state == 0:
        return 0
    else:
        return ((p / p_star)**arrivals) * (((1 - p) / (1 - p_star))**completions)
```

In [118]:
```
#do the same thing for importance sampling as we did before for Monte-Carlo
c3 = 3
c5 = 5
c7 = 7
counter3 = 0
counter5 = 0
counter7 = 0
iterations = 100000
for i in range(0, iterations):
    counter3 += IS(c3)
    counter5 += IS(c5)
    counter7 += IS(c7)

print("Probability of queue hitting 3: " + str(counter3 / iterations))
print("Probability of queue hitting 5: " + str(counter5 / iterations))
print("Probability of queue hitting 7: " + str(counter7 / iterations))
```

```
Probability of queue hitting 3: 0.17389056000005912
Probability of queue hitting 5: 0.08823111680004211
Probability of queue hitting 7: 0.05041343692799139
```

This one also seems to be OK.

Comparing variances

Let's simulate 100 different paths to give us an estimate on the γ and do that 100 times and calculate the variance in both regular Monte-Carlo and importance sampling.

In [76]:
```
#functions for estimating the probability of hitting certain queue length for both functions
def estimateMC(c, iterations):
    runs = []
    for i in range(0, iterations):
        runs.append(path(c))
    return np.mean(runs)

def estimateIS(c, iterations):
    runs = []
    for i in range(0, iterations):
        runs.append(IS(c))
    return np.mean(runs)
```

In [125]:
```
iterations = 100
estimation_iterations = 100
#make numpy arrays to store the results
moca = np.empty((iterations, 3), dtype = float)
imsa = np.empty((iterations, 3), dtype = float)

#calculate estimates for different queue lengths and save the results
for i in range(0, iterations):
    for j in range(0,3):
        moca[i, j] = estimateMC(3 + 2*j, estimation_iterations)
        imsa[i, j] = estimateIS(3 + 2*j, estimation_iterations)

#calculate the means and standard deviations of estimates from both methods
mc_mean = np.mean(moca, 0)
mc_std = np.std(moca, 0)
is_mean = np.mean(imsa, 0)
is_std = np.std(imsa, 0)
```

In [126]:
```
print("{:13}{:9}{:9}{:9}".format("","c = 3", "c = 5", "c = 7"))
print("{:10}{:9.4f}{:9.4f}{:9.4f}".format("Mean MC:", mc_mean[0], mc_mean[1], mc_mean[2]))
print("{:10}{:9.4f}{:9.4f}{:9.4f}".format("Mean IS:", is_mean[0], is_mean[1], is_mean[2]))
print("{:10}{:9.4f}{:9.4f}{:9.4f}".format("Std MC:", mc_std[0], mc_std[1], mc_std[2]))
print("{:10}{:9.4f}{:9.4f}{:9.4f}".format("Std IS:", is_std[0], is_std[1], is_std[2]))
```

```
             c = 3    c = 5    c = 7
Mean MC:     0.1706   0.0821   0.0479
Mean IS:     0.1709   0.0899   0.0500
Std MC:      0.0408   0.0281   0.0209
Std IS:      0.0234   0.0166   0.0089
```

We can see that importance sampling gives us notably smaller standard deviations in all of the cases but the effect is especially clear in the case of $c = 7$ where the occurrance of the event is much lower. There importance sampling has a standard deviation that is over 2 times smaller than the basic Monte-Carlo equivelant. So again we see that it is possible to reduce the deviation in simulation results for rare events by selecting a smarter approach than just brute force Monte-Carlo.