Homewok 1 ¶

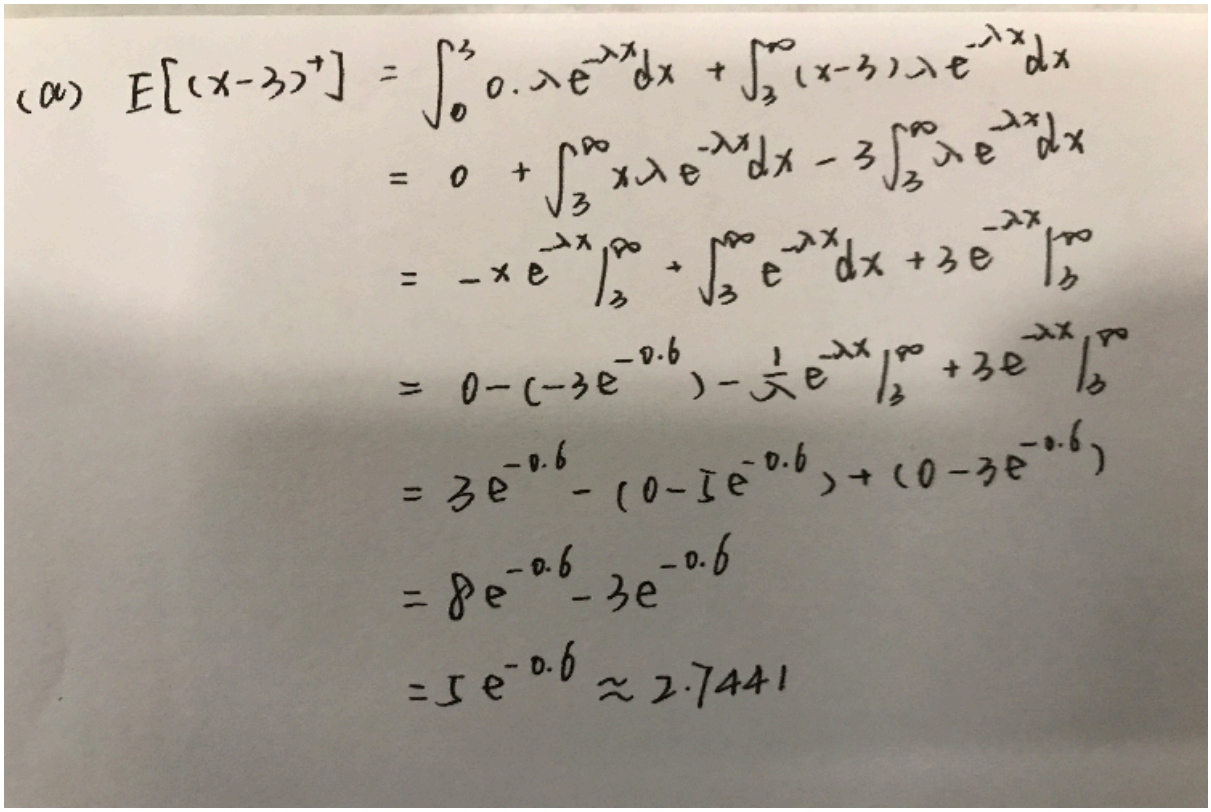
[redacted]

In [108]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import display
from PIL import Image
```

Problem 1

In [146]:
```
path1 = '/Users/baronma/Downloads/Problem_1_a.jpeg'
img1 = Image.open(path1)
width, height = img1.size
size1 = width/2, height/2
img1.resize(size)
```

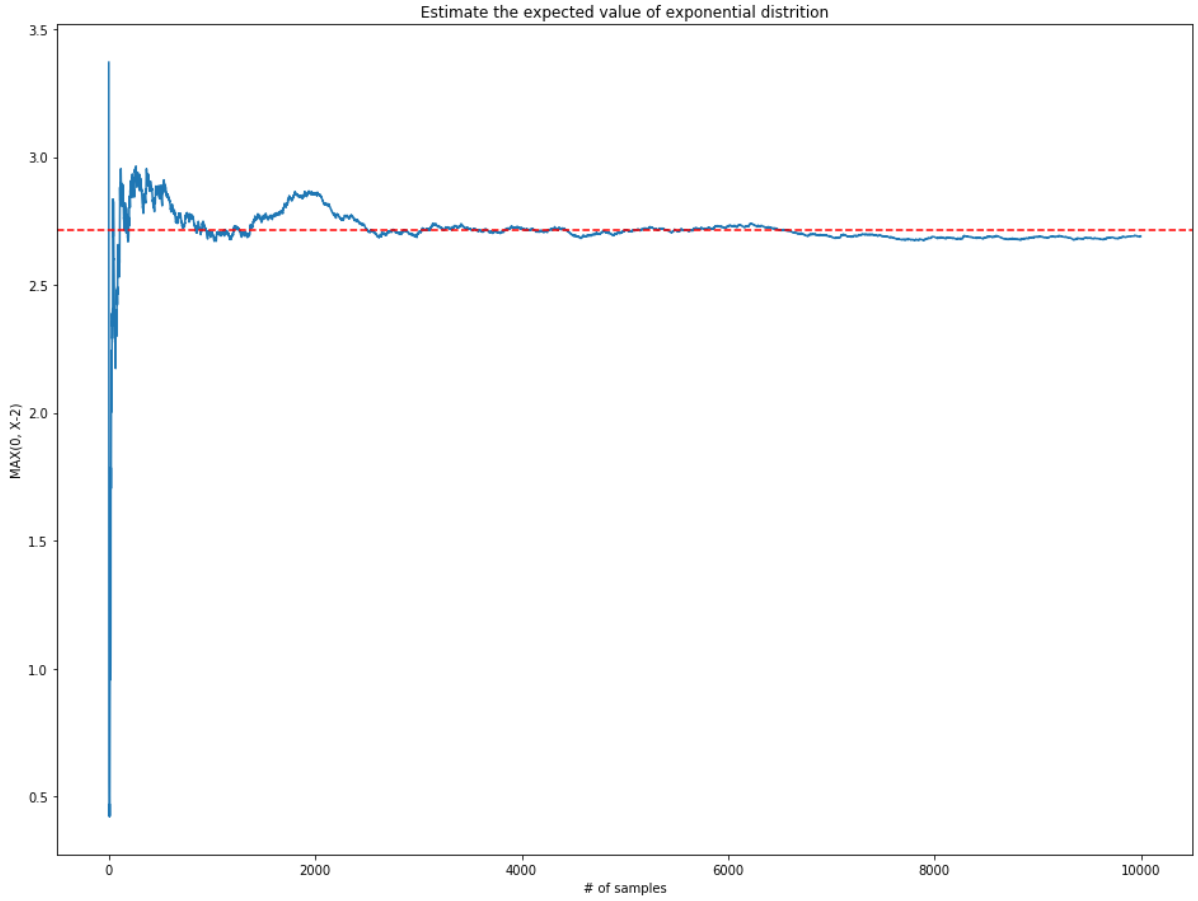
Out[146]:



In [152]:
```
np.random.seed(1)
data = np.random.exponential(scale=1/0.2, size=10000)

y = [max(x-3, 0) for x in data]

#return cumulative sums from list ex
fsum = np.cumsum(y)

expected_v = []
for i in range(1, 10000):
    expected_v.append(fsum[i]/i)

mean = np.mean(expected_v)
StD = np.std(expected_v,ddof=1) #here specifies ddof = 1 gives an unbi
ased estimator (using N-1)

Upper = mean + 1.96*StD/np.sqrt(len(expected_v)) #upper 95% CI
Lower = mean - 1.96*StD/np.sqrt(len(expected_v)) #lower 95% CI
print "95% CI for the expected value is between", '%.4f' %Lower, "and"
, '%.4f' %Upper
print "the convergence mean of:", '%.4f' %mean
```

```
95% CI for the expected value is between 2.7139 and 2.7179
the convergence mean of: 2.7159
```

In [151]:
```
#plot the averages showing convergence
plt.figure(figsize = (16,12))
plt.plot(expected_v)
plt.axhline(y=mean, color='r', linestyle='--')
plt.xlabel('# of samples')
plt.ylabel('MAX(0, X-2)')
plt.title("Estimate the expected value of exponential distrition")
plt.show()
```



Problem 2

In [97]:
```
def Failure ():
    global S
    global Slast
    global Tlast
    global Area
    global NextFailure
    global NextRepair

    S = S - 1
    if S == 1:
        NextRepair = clock + 2.5
        NextFailure = clock + np.ceil(6*np.random.random())
    # Update the area under the sample path and the
    # time and state at the last event
    Area = Area + (clock - Tlast)* Slast
    Tlast = clock
    Slast = S
```

In [98]:
```
def Repair():
    global S
    global Slast
    global Tlast
    global Area
    global NextFailure
    global NextRepair

    S = S + 1
    if S == 1:
        NextRepair = clock + 2.5
        NextFailure = clock + np.ceil(6*np.random.random())

    #NextRepair = float('inf')
    Area = Area + Slast * (clock - Tlast)
    Slast = S
    Tlast = clock
```

In [99]:
```
def Timer():
    global clock
    global NextRepair
    global NextFailure
    global NextEndSimulation

    if (NextFailure < NextRepair) & (NextFailure < NextEndSimulation):
        result = "Failure"
        clock = NextFailure
        NextFailure = float('inf')

    else:
        if NextRepair < NextEndSimulation:
            result = "Repair"
            clock = NextRepair
            NextRepair = float('inf')

        else:
            result = "EndSimulation"
            clock = NextEndSimulation
            NextEndSimulation = float('inf')
    return result
```

In [100]:
```
def EndSimulation():
    global Area
    Area = Area + Slast * (clock - Tlast)
```

(1) Generate the average number of functional components until time T = 1000

In [102]:
```
N = 100
# Define lists to keep samples of the outputs across replications
TTF_list = []
Ave_list = []
# fix random number seed
np.random.seed(1)

for reps in range(0,N):
    # start with 2 functioning components at time 0
    clock = 0
    S = 2
    # initialize the time of events
    NextRepair = float('inf')
    NextFailure = np.ceil(6*np.random.random())
    NextEndSimulation = 1000
    # Define variables to keep the area under the sample path
    # and the time and state of the last event
    Area = 0.0
    Tlast = 0
    Slast = 2
    NextEvent = ''

    while NextEvent != "EndSimulation": # While system is functional
        NextEvent = Timer()

        if NextEvent == "Repair":
            Repair()
        else:
            if NextEvent == "Failure":
                Failure()
            else:
                EndSimulation()

    # add samples to the lists
    # TTF_list.append(clock)
    Ave_list.append(Area/clock)

# print sample averages
print 'Estimated expected ave. # of func. comp. till T:', np.mean(Ave_
list)
```

```
Estimated expected ave. # of func. comp. till T: 1.2646849999999998
```

(2) Generate the average number of functional components until time T = 2000

In [103]:
```
N = 100
# Define lists to keep samples of the outputs across replications
TTF_list = []
Ave_list = []
# fix random number seed
np.random.seed(1)

for reps in range(0,N):
    # start with 2 functioning components at time 0
    clock = 0
    S = 2
    # initialize the time of events
    NextRepair = float('inf')
    NextFailure = np.ceil(6*np.random.random())
    NextEndSimulation = 2000
    # Define variables to keep the area under the sample path
    # and the time and state of the last event
    Area = 0.0
    Tlast = 0
    Slast = 2
    NextEvent = ''

    while NextEvent != "EndSimulation": # While system is functional
        NextEvent = Timer()

        if NextEvent == "Repair":
            Repair()
        else:
            if NextEvent == "Failure":
                Failure()
            else:
                EndSimulation()

    # add samples to the lists
    # TTF_list.append(clock)
    Ave_list.append(Area/clock)

# print sample averages
print 'Estimated expected ave. # of func. comp. till T:', np.mean(Ave_
list)
```

```
Estimated expected ave. # of func. comp. till T: 1.2615124999999998
```

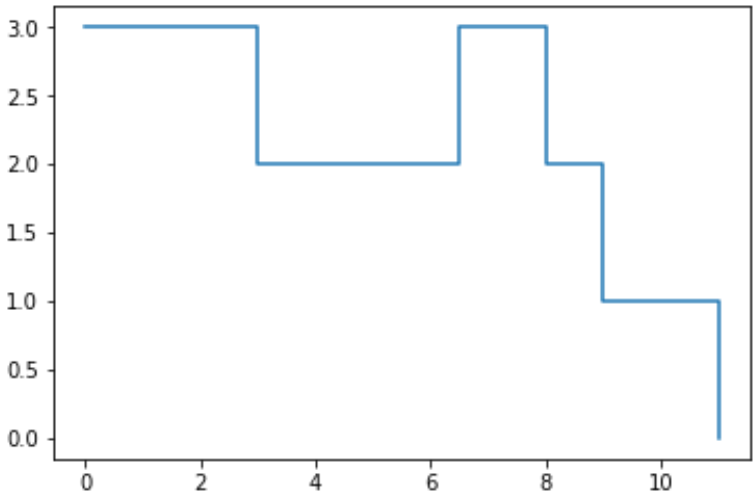
From the two results, we can see obvierously that the average number of functional components until time T = 2000 are almost the same as those until time T = 1000, but still has slightly difference.
That means that as the T increases, the average number of functional components will converge.

Problem 3

Gererate one sample path and $S(t)$ and computes the time to failure.

In [18]:
```
# start with 2 functioning components at time 0
clock = 0
S = 3

# fix random number seed
np.random.seed(1)

# initialize the time of events
NextRepair = float('inf')
NextFailure = np.ceil(6*np.random.random())
# lists to keep the event times and the states
EventTimes = [0]
States = [3]

while S > 0:
    # advance the time
    clock = min(NextRepair, NextFailure)

    if NextRepair <= NextFailure:
        # next event is completion of a repair
        S = S + 1
        if S == 2:
            NextRepair = clock + 3.5
        if S == 3:
            NextRepair = float('inf')
    else:
        # next event is a failure
        S = S - 1
        if S == 2:
            NextRepair = clock + 3.5
            NextFailure = clock + np.ceil(6*np.random.random())
        if S == 1:
            NextFailure = clock + np.ceil(6*np.random.random())

    # save the time and state
    EventTimes.append(clock)
    States.append(S)

# plot the sample path
print 'TTF was:', clock
plt.plot(EventTimes, States, drawstyle = 'steps-post')
plt.show()
```

```
TTF was: 11.0
```



Run multiple replications and estimate the expected value of time to failure average number of fucntioning components till failure.

In [137]:
```
# Set number of replications
N = 1000
# Define lists to keep samples of the outputs across replications
TTF_list = []
Ave_list = []

# fix random number seed
np.random.seed(1)

for rep in range (0,N):
    # start with 2 functioning components at time 0
    clock = 0
    S = 3
    # initialize the time of events
    NextRepair = float('inf')
    NextFailure = np.ceil(6*np.random.random())
    EventTimes = [0]
    States = [3]
    # Define variables to keep the area under the sample path
    # and the time and state of the last event
    Area = 0.0
    Tlast = 0
    Slast = 3

    while S > 0:
        # advance the time
```

```
        clock = min(NextRepair, NextFailure)

        if NextRepair < NextFailure:
            # next event is completion of a repair
            S = S + 1
            if S == 2:
                NextRepair = clock + 3.5
            if S == 3:
                NextRepair = float('inf')
        else:
            # next event is a failure
            S = S - 1
            if S == 2:
                NextRepair = clock + 3.5
                NextFailure = clock + np.ceil(6*np.random.random())
            if S == 1:
                NextFailure = clock + np.ceil(6*np.random.random())
        # Update the area under the sample path and the
        # time and state of the last event
        Area = Area + (clock - Tlast)* Slast
        Tlast = clock
        Slast = S

    # save the TTF and average # of func. components
    TTF_list.append(clock)
    Ave_list.append(Area/clock)

print 'Estimated expected TTF:', np.mean(TTF_list)
print 'Estimated expected ave. # of func. comp. till failure:', np.mea
n(Ave_list)
```

```
Estimated expected TTF: 35.312
Estimated expected ave. # of func. comp. till failure: 2.05510454562
63872
```

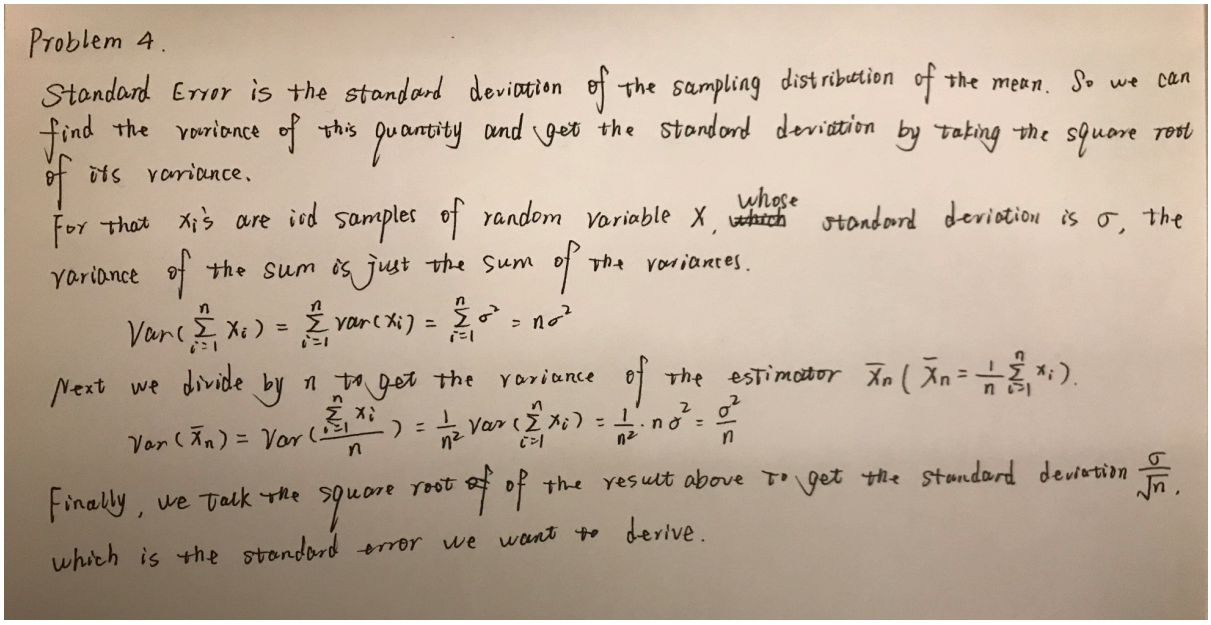
In [150]:
```
StandD = np.std(TTF_list,ddof=1)
UpperCI = np.mean(TTF_list) + 1.96*StandD/np.sqrt(len(TTF_list)) #uppe
r 95% CI
LowerCI = np.mean(TTF_list) - 1.96*StandD/np.sqrt(len(TTF_list)) #lowe
r 95% CI
print "95% CI for the expected time to failure is between", '%.4f' %Lo
werCI, "and", '%.4f' %UpperCI
```

```
95% CI for the expected time to failure is between 33.4496 and 37.17
44
```

Problem 4

In [124]:
```
path2 = '/Users/baronma/Downloads/Problem_4.jpg'
img2 = Image.open(path2)
width, height = img2.size
size2 = width/2, height/2
img2.resize(size2)
```

Out[124]:



Based on the solution above, I used the Standard Normal Distribution to proof my result. Every time generate n noraml distribution random numbers and get the mean of these numbers, do this k times in a for loop and get the distribution of mean. Then calculate the standard deviation of the mean distribution, which should be equal to the result above.

In [139]:
```
k = 100000; # Number of Samples.
n = 1000; # Number of random numbers generated each time.
# Standard Normal Distribution.
mu = 0;
sigma = 1;
```

In [140]:
```
normal = []
normalmean = []
stand = 0
np.random.seed(1)
for i in range(0, k):
    norm = np.random.normal(mu, sigma, n)
    normalmean.append(np.mean(norm))

standardD = np.std(normalmean, ddof = 1)
standardD
```

Out[140]: 0.031551712374224085

In [144]:
```
# Standard Error.
sigma/np.sqrt(n)
```

Out[144]: 0.03162277660168379

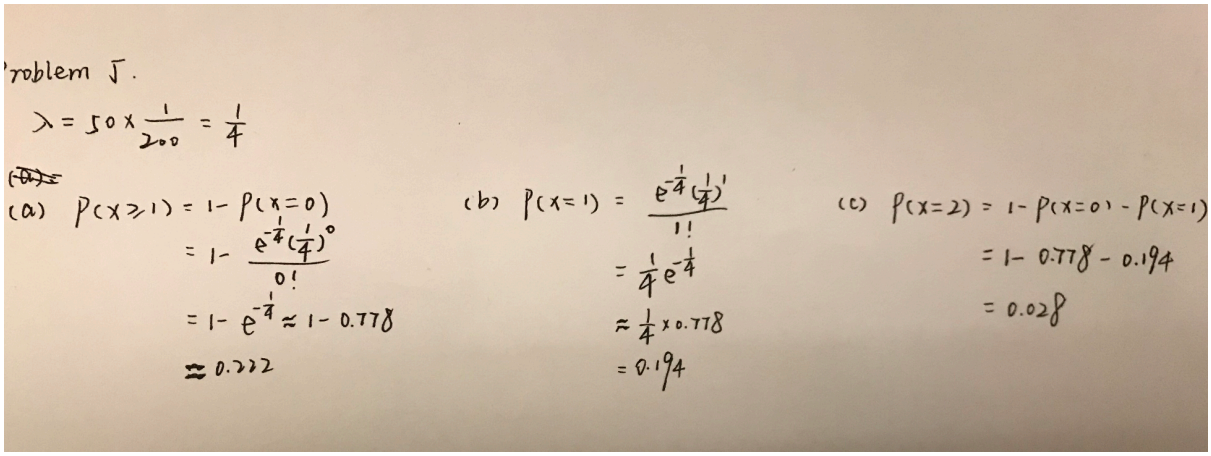
After the simulation, the se of the estimator Xn_bar is sigma/sqrt(n)

Problem 5

We could use the binomial model for an exact answer, or the Poisson model for an approximate answer. Since the problem asks for an approximation, we use the Poisson.

In [125]:
```
path3 = '/Users/baronma/Downloads/Problem_5.jpg'
img3 = Image.open(path3)
width, height = img3.size
size3 = width/2, height/2
img3.resize(size3)
```

Out[125]:



We can also use simulation method to generate poisson random numbers to derive the approximately results.

In [48]:
```
np.random.seed(1)
def poisson(lam, k):
    x = np.random.poisson(lam, k)
    return x
```

(a)

In [70]:
```
f = 10000
lam = 0.25
lot = poisson(lam, f)
```

In [82]:
```
s1 = 0
for j in lot:
    if j == 0:
        s1 += 1
p1 = float(s1)/f
```

In [89]:
```
print "The possibility that you will win a prize at least once is", (1 - p1)
```

```
The possibility that you will win a prize at least once is 0.2237
```

(b)

In [79]:
```
s2 = 0
for j in lot:
    if j == 1:
        s2 += 1
p2 = float(s2)/f
```

In [90]:
```
print "The possibility that you will win a prize exactly once is", p2
```

```
The possibility that you will win a prize exactly once is 0.1946
```

(c)

In [91]:
```
print "The possibility that you will win a prize at least twice is",(1
- p1 - p2)
```

```
The possibility that you will win a prize at least twice is 0.0291
```