Final Assignment - CSC 521 By Zhenyang Lu

0. Part one

0.0 What are the characteristics of a good random number generator?

Periodicity, a good number generator should have a periodicity that is long enough to produce non-cyclic random number.

Unpredictability, a good number generator cannot be told what`s the next number. This is also called uncorrelated between numbers.

Besides "random enough", you might want to pursue a speed of a random number generator in order to keep your simulation as fast as possible.

0.1What is the difference between a Poisson distribution and the exponential distribution?

Poisson distribution and exponential distribution are related in that they two have the same parameter, lambda.

Poisson distribution, compared to exponential distribution, tells you that given a certain rate, how many times the event occurs at a fixed period of time, say the events are independent.

While exponential distribution tells you that given a rate that the event occurs in a fixed time what is the time interval between the events.

0.2 What are the main differences between the Gaussian distribution and the Pareto distribution?

Gaussian distribution is a bell-shape and systematically distributed density function. It is often used as a rough approximation to describe daily business that cluster around a single mean. Gaussian distribution is good to describe daily business gain or loss, since the numbers you will get are fairly around certain range, fluctuating less than that of Pareto distribution.

Pareto distribution is an exponentially distributed density function, the possibility of extreme values grow very rapidly. It is used to describe ' unusual things' small possibility that a huge number happens such as how much you will loss in a financial crisis or how much you will hit the lottery, in which odds are very small, but loss or gain are very large.

0.3 When pricing an option, when do you need to use Monte Carlo?

Options are financial tools for managing potential risks of underlying assets. The price of an option is dependent on the risk of the underlying assets, which are usually currencies, stocks or commodities. Since the prices of these assets are volatile, the prices of their options are also volatile. Thus, we will have to use Monte Carlo, under some assumptions, to simulate the daily return for the option in order to see the potential risk for the options, which is the base for pricing the options.

0.4 What are the two ways you know to compute the statistical error on an average?

CSC 521 has told two ways to do it.

First, you calculate the mean and square of mean of your data and use $\mathrm{var}(X) = E[(X - E\ [X])^2] = E[X^2] - E^2[X]$ to get your variance. Then under the assumption that your data are normally distributed, you could estimate your statistical error by standard deviation.

Second, you could use Monte Carlo algorithm to simulate many times your experiment and store your results, then use bootstrap algorithm to compute the means of every spectrums and sort this result, finding the 16th and 84th percentile. The difference between the 16th or 84th percentile and the mean of means is a estimator of standard deviation.

0.5 What is the meaning and purpose of “resampling"?

Resampling means that estimating the parameters (such as mean and variance) of a sample by using a subset of available data or drawing randomly with replacement from a set of data points.

The benefit of using resampling is that it can be used in any kind of data, no matter what is the distribution of the data. Thus, we do not have to make assumptions about what is the "shape" of the data.

Another advantage is that if we have to do the process of generating random numbers for many times in our Monte Carlo simulation, we could simply generate them and store them in a list, then just use it by resampling from the list, rather than make generate random numbers when we need to. By doing this, we could speed up our computation.

1. Part two

You run a web service that performs computations on behalf of your users. When a request arrives to your computers, you process it and you deliver an answer. Requests arrive at random intervals with an average of one per minute. Once a request arrives, if you have a computer server available to process it, you assign it and it will complete in a random time given by the Pareto distribution with alpha = 2.5 and X_m = 5, unit in minutes. If the request arrives and all your servers are busy it will be queued. Each server can deal with one request at the time. One request can be queued but for no longer than 5minutes. If a request is queued for more than 5 minutes it is dropped. You will have to pay $10 in penalty for each dropped request.

1.0 Analysis

First, let me construct a class that simulate this web service company called **class webservice()** and define random Pareto random number generator, called **Pareto()**, inside the class:

```
import random
from numeric import *

class webservice():
    time=30*24*60  #assume one month, and 30 days per month
    server = []    # how many servers
    cost = 0.0
    queue_mission =[]  # contains the mission on the queue.
    queue_time = [] # contains the waiting time of each mission on queue


    #a pareto PRNG
    def pareto(self,xm=5,alpha=2.5):
        return float(xm)*(1.-random.random())**(-1.0/alpha)
```

In the previous code, I am simulating the cost for a month rather than a year to speed up the program. By creating two lists, **queue_mission=[]** and **queue_time=[],** I am assuming that these two lists act like a two-dimension array, in which the first dimension contains the mission, the second contains the waiting time for the mission.

What does "mission" mean in my code? Every mission that is assigned to the **queue_mission=[]** is a Pareto random number with alpha = 2.5 and X_m = 5, unit in minutes. When "the mission" is passed to the **server=[]** list, the "mission" can be directly subtracted by how many minutes it has taken(**for simulate_once()**)in this case, I simulate 43200 minutes, length of a month).

After creating lists of server, **queue_time** and **queue_mission**, I define two methods in the class that imitate behaviors of putting a mission in the queue and putting a mission from the queue into the server by the following code:

```
def put_on_queue(self,x): # def a method to put next mission on queue
    for i in range(len(self.queue_mission)):
        if self.queue_mission[i] == 0:
            self.queue_mission[i] = x
            # return the index of queue where the mission should be placed.
            self.queue_time[i]= .0000001
            # when the mission has been put on queue, the timer for it should be 0
            # for convenience of timing it after, it is 0.0000001 rather than 0
            break

def put_on_server(self):
    #def a method put the first mission on queue onto server.
    for i in range(len(self.server)):
        if self.server[i] == 0.0:
            if self.queue_mission[0]>0:
                self.server[i] = self.queue_mission[0]
                # put the mission on server.
                self.queue_time.remove(self.queue_time[0])
                # First in First out
                self.queue_time.append(0.00000)
                self.queue_mission.remove(self.queue_mission[0])
                self.queue_mission.append(0.00000)
```

The **put_on_queue()** method checks which spot on the queue is empty and then put the mission from client onto the queue.

The **put_on_server()** method checks which server is available by testing which server is "zero", which I assume is the sign of "available" and import the mission from the queue to the server.

Finally, I still have to define two methods to calculate the waiting time for missions on servers and queue. These two behaviors can be done by code:

```
def timer_queue(self,t):
# def a method that count the waiting time for each mission on queue
    for i in range(len(self.queue_time)):
        if self.queue_time[i] >0:
            self.queue_time[i] = self.queue_time[i]+t

def timer_server(self,t):
    # def a method to manage the time
    for i in range(len(self.server)):
        self.server[i] = max(self.server[i]-t,0)
```

Every minute I simulate, the mission on server will be reduced by one and the mission on queue should be added by one. However, I should point out that since the initial numbers of **queue_time=[]** are zero, I should only add one to those that larger than zero rather than adding one to each element on **queue_time[].** Also, the mission on server cannot be lower than zero, which indicates that the mission has been solved, so the minimum for each element on server is zero.

Till now, I have constructed the main code to behave this web service company. All the methods in the **webservice()** class are centered on the three lists, **queue_time, queue_mission** and **server**. They all do certain things to the three lists.

1.1 Simulate_once()

By doing **simulate_once(),** I want to simulate a month's cost. This can be done by code:

```
def simulate_once(self,N):
    # N = number of servers, simulate a month.
    self.cost = N * 25
    # 25 is from 300/12, monthly cost for each server
    self.queue_time = [.0]*100
    self.queue_mission = [.0]*100
    self.server = [0.0 for i in range(N)]# how many servers
    t = 0.0
    p =0.0
    for i in range(self.time):
        t = t + random.expovariate(1.0)
        while t>1:
            self.put_on_queue(self.pareto())
            t = t - 1
        p = self.drop_mission()
        while p>0:
            self.cost = self.cost+p
            p = self.drop_mission()
        self.put_on_server()
        self.timer_server(1)
        self.timer_queue(1)
    return self.cost/N
```

The argument, **N**, that **simulate_once()** takes is how many servers you want to put into simulation. The lines before the **for** loop are initializations for the global parameters of the class. The first **while** loop pass the upcoming mission(s) onto the queue **(queue_mission=[]**). The exponential variable **t** determines how many missions has/have come, when **t** is larger than 1, it comes one mission, when **t** is larger than 2, it comes two missions and etc., The second **While** loop indicates how many mission(s) on the queue has/have been waiting for 5 minutes or more and how much is the dropping cost, which has to be added to the total cost (global parameter of **self.cost**). This **simulate_once()** method returns the average cost for each server if the total number of servers is **N.**

By simulating different numbers of servers, we can get the inference of how many servers is ideal in order to minimize the cost. The result is like (ignore the client code for realizing the result):

```
average cost per server of 1 server(s) is 382995.666667
average cost per server of 3 server(s) is 95446.2222222
average cost per server of 6 server(s) is 24085.9444444
average cost per server of 9 server(s) is 4212.14814815
average cost per server of 12 server(s) is 367.638888889
average cost per server of 15 server(s) is 40.9777777778
average cost per server of 20 server(s) is 25.1166666667
average cost per server of 25 server(s) is 25.0
>>>
```

From this, we can see that when the number of servers comes to 20-25, the average cost per server is 25, which is the cost for only the fixed cost for each server. One thing I should point out is that since the result of **simulate_once()** fluctuates a lot, there may be chances that the average cost per server occurs 25 when **N** is around 15, which indicates that sometimes 15 servers are enough to maximize your profit. Due to the slowness of running the **simulate_once()** method, I will not provide all the data to prove this conclusion. This can be shown by the following list, which contains data for **simulate_once(N)** with a parameter of 15:

```
>>> print list
[46.333333333333336, 50.333333333333336, 47.0, 45.0, 39.666666666666664, 56.333333333333336, 39.666666666666664, 37.666666666666664, 25.666666666666668, 40.333333333333336, 48.333333333333336, 43.666666666666664, 44.333333333333336, 39.666666666666664, 32.333333333333336, 33.0, 41.666666666666664, 37.666666666666664, 35.666666666666664, 35.0, 42.333333333333336, 34.333333333333336, 41.666666666666664, 43.0, 37.666666666666664, 30.333333333333332, 41.666666666666664, 39.0, 39.0, 35.0, 49.666666666666664, 37.666666666666664, 34.333333333333336, 39.666666666666664, 41.666666666666664, 42.333333333333336, 39.666666666666664, 35.666666666666664, 55.0, 42.333333333333336, 44.333333333333336, 38.333333333333336, 30.333333333333332, 39.0, 43.0, 45.666666666666664, 42.333333333333336, 43.666666666666664, 43.0, 37.0, 49.0, 46.333333333333336, 40.333333333333336, 43.666666666666664, 35.666666666666664, 56.333333333333336, 49.0, 35.666666666666664, 40.333333333333336, 43.0, 53.0, 48.333333333333336, 45.666666666666664, 40.333333333333336, 47.0, 40.333333333333336, 43.0, 44.333333333333336, 42.333333333333336, 47.666666666666664, 39.0, 39.666666666666664, 41.666666666666664, 46.333333333333336, 38.333333333333336, 39.0, 40.333333333333336, 31.666666666666668, 54.333333333333336, 41.0, 41.0, 39.666666666666664, 4
, 41.0, 36.333333333333336, 47.0, 40.333333333333336, 45.666666666666664, 49.666666666666664, 39.0, 43.666666666666664, 40.333333333333336, 40.333333333333336, 48.333333333333336, 43.666666666666664]
```

You could vaguely see there exists 25 on the first line. Also, there will be odds that 25, which is the lowest average cost in this simulation, happens when **N** is lower than 15.

1.2 95% VaR less than $10,000

Since I have already got the result of **simulate_once()** with argument **N** ranging from 1 to 26, I could simply solve this problem by enumerating **N** starting from 9 to 20 in order to get the cutting number of **N** such that 95% VaR is less than $10,000 rather than make a method taking **N** as argument and return the 95% VaR. Since the following graph, which we get in 1.1 section,

```
average cost per server of 1 server(s) is 382995.666667
average cost per server of 3 server(s) is 95446.2222222
average cost per server of 6 server(s) is 24085.9444444
average cost per server of 9 server(s) is 4212.14814815
average cost per server of 12 server(s) is 367.638888889
average cost per server of 15 server(s) is 40.9777777778
average cost per server of 20 server(s) is 25.1166666667
average cost per server of 26 server(s) is 25.0
>>>
```

has shown that when **N** is 9, total cost is 9*4212, 37980, and when **N** is 12, total cost is 12*367, 4404, I **assume** that the **N** that makes 95% VaR less than $10,000 is possibly between 9 - 12 and above 400. However, for the second case, say **N** is 400, although the total cost is 400*25 (25 is monthly cost for each server), your total profit is certainly positive when you take how much the client pay for each mission in consideration. So I will **presumably ignore** this case.

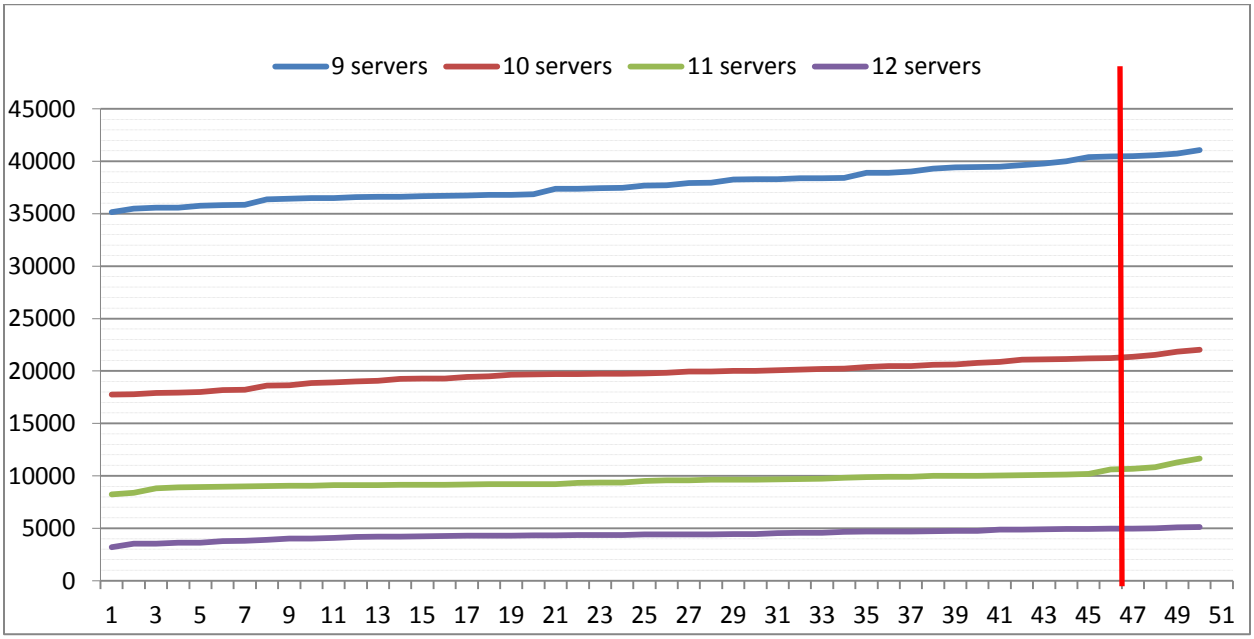
To speed up my program, for each **N**, I simulate 50 times and store and sort the data. The code for this part is like:

```
list_10= []
list_11 = []
list_12 = []
for i in range(50):
    list_9.append(x.simulate_once(9))
for i in range(50):
    list_10.append(x.simulate_once(10))
for i in range(50):
    list_11.append(x.simulate_once(11))
for i in range(50):
    list_12.append(x.simulate_once(12))
list_9.sort()
list_10.sort()
list_11.sort()
list_12.sort()
print "95% VaR for 9 servers is", list_9[int(.95*50)]
print "95% VaR for 10 servers is", list_10[int(.95*50)]
print "95% VaR for 11 servers is", list_11[int(.95*50)]
print "95% VaR for 12 servers is", list_12[int(.95*50)]
```

Result of this code:

```
warning: matplotlib not available
95% VaR for 9 servers is 40805.0
95% VaR for 10 servers is 22120.0
95% VaR for 11 servers is 10785.0
95% VaR for 12 servers is 5170.0
```

To further infer the data, I plot the lists for 9 -12 servers. We could see the following lines plot:



The vertical axis is loss and the red line is 95% VaR. Combined with the two graphs above, we could see that if you have 12 servers, it is definitely you will have a 95% VaR less than $10,000.

However, since this experiment has only been done for 50 times, if you expand the times of your simulation for N equal to 11, you will also be likely to have a 95% VaR that is less than $10,000. From the lines plot we could see that the 45th data of the sorted list, shown by green line, is around $10,000. This further proves that 11 servers are also able to satisfy a 95% VaR which is less than $10,000, definitely not a very solid conclusion.

1.3 Breakeven Point for profit

Since for different number of servers, **N**, the price for breakeven points is different, I should test different prices for each **N.** That is how I do this section.

First, let define a method that take **N** and **price** as arguments, **price** means what is the price for each mission assigned. **N** is still number of traders. Code is like:

```
def profit(self,N,price):
    # def a method to get the profit, N is number of servers, price is the price for each mission finished.
        self.simulate_once(N)
        profit =(self.mission_assigned-(self.cost-25*N)/10)*price - self.cost
        print "cost=",self.cost,"profit=",profit
```

Then I just use this method to test the breakeven point with such codes as:

```
for i in range(75,86):
    x.profit(1,i)

for i in range(30,40):
    x.profit(2,i)

for i in range(15,25):
    x.profit(3,i)
```

The result is like (non-exhausted):

```
when price is 75 , number of servers is 1 , profit is -17857.7152241
when price is 76 , number of servers is 1 , profit is -12991.4420046
when price is 77 , number of servers is 1 , profit is -8215.34394698
when price is 78 , number of servers is 1 , profit is 3188.8099806
when price is 79 , number of servers is 1 , profit is -1036.0739474
when price is 80 , number of servers is 1 , profit is 10941.6781333
when price is 81 , number of servers is 1 , profit is 9966.26815755
when price is 82 , number of servers is 1 , profit is 16598.0470101
when price is 83 , number of servers is 1 , profit is 3008.00347849
when price is 84 , number of servers is 1 , profit is 19675.1608586
when price is 85 , number of servers is 1 , profit is 27794.44941
```

```
when price is 30 , number of servers is 2 , profit is -41504.0969613
when price is 31 , number of servers is 2 , profit is -41761.9407919
when price is 32 , number of servers is 2 , profit is -19160.5400138
when price is 33 , number of servers is 2 , profit is -19205.7741396
when price is 34 , number of servers is 2 , profit is 1483.56448669
when price is 35 , number of servers is 2 , profit is 9444.49301405
when price is 36 , number of servers is 2 , profit is 17805.3134766
when price is 37 , number of servers is 2 , profit is 28373.3497514
when price is 38 , number of servers is 2 , profit is 37098.2311308
when price is 39 , number of servers is 2 , profit is 45760.8067883
```

and

```
when price is 5 , number of servers is 6 , profit is 3105.63697037
when price is 6 , number of servers is 6 , profit is 32814.7091178
when price is 7 , number of servers is 6 , profit is 59406.5420375
when price is 1 , number of servers is 7 , profit is -69190.4303637
when price is 2 , number of servers is 7 , profit is -31383.7581723
when price is 3 , number of servers is 7 , profit is -5195.63216816
when price is 1 , number of servers is 8 , profit is -28773.5294341
when price is 2 , number of servers is 8 , profit is 9112.26222595
when price is 0.5 , number of servers is 9 , profit is -20479.7877737
when price is 1.0 , number of servers is 9 , profit is -815.211454026
when price is 0.2 , number of servers is 10 , profit is -12918.1827362
when price is 0.3 , number of servers is 10 , profit is -5845.31819485
when price is 0.4 , number of servers is 10 , profit is -4289.2612595
when price is 0.5 , number of servers is 10 , profit is -1536.59546298
when price is 0.6 , number of servers is 10 , profit is 5661.56194972
when price is 0.7 , number of servers is 10 , profit is 8079.46561226
```

From these data, you can record which prices makes the profit larger than zero, then I just compile these data, the graph of breakeven points is like (using excel):

Unit = dollar	
# of servers	breakeven price
1	78~79
2	33~34
3	19~20
4	12~13
5	7~8
6	4~5
7	3~4
8	1~2
9	1.0~1.5
10	0.5~0.6
11	0.25~0.35
12	0.05~0.15
13	0.04~0.08
14	0.01~0.03
15	0.01~0.02
16	0.01~0.02
17	0.015~0.02
18	0.01~0.015
19	0.01~0.015
20	0.01~0.015

Even if you continue increasing number of traders, which is an argument for the code, the breakeven point cannot be higher. Since the total number of mission is fixed. Your total profit cannot be increased by adding the number of servers.

1.4 computing average profit using bootstrap algorithm

I **assume** that the best number of servers, **N**, is 20 and price for each mission is 0.1 as an example. Then I just compute the average profit ranging from the mean minus standard deviation and the mean plus standard deviation$(\mu - \sigma, \mu + \sigma)$. The code for this part of answer is like:

```
x = webservice()

list = []

for i in range(100): # create a list contains profit for 20 servers and price of 0.01
    list.append(x.profit(20,0.1))

def resample(list,n):
    # random resample for the list created and return a subset of the list
    return [random.choice(list) for i in range(n)]

def bootstrap(list,nboot=100):
    #bootstrap algorithm which returns μ-σ and μ+σ
    mu = sum(list)/len(list)
    mu_samples = [ ]
    for i in range(nboot):
        sample = resample(list,len(list))
        mu_sample = sum(sample)/len(sample)
        mu_samples.append(mu_sample)
    mu_samples.sort()
    print "sigma-mu=",mu_samples[15], "mu=",mu ,"sigma+mu=",mu_samples[100-16]
```

The result is like:

```
>>> bootstrap(list)
sigma-mu= 3813.36740482 mu= 3816.33695023 sigma+mu= 3817.78208755
>>>
```

Sigma is around 1~3.

1.5 Business discussion

Introduction: This is a really good venture investment project. In this case, each server cost you $25 per month. You receive mission from client and solve it using your server. Expectedly, you will receive around 1400-1500 missions per day and each mission cost a server around 9 minutes to solve. If a mission cannot be solved within 5 minutes, you will loss $10 for each you dropped.

Potential risk: if the number of servers is lower, you will possibly loss money because those numbers of servers cannot deal with so many mission, so you probably losing money by dropping mission and accusing penalty. If you want to keep your potential risk and do not want to bankrupt with a confidence of 95% that do not loss money by $10,000, you will have to at least rent 12 servers.

Promising benefits: according to my analysis above, the potential profit for this project is very high. If you could rent around 15-20 servers, your average cost for each server is just the rent you pay. And if your client agreed with a price of $0.02 per mission you done, you will almost never loss your money. Since the breakeven price for you to balance your cost with your profit is $0.015~0.02 per mission. Let me give you a brief example of how much money you will earn, assume you charge $0.1 per mission you solved and you have 20 servers, your total profit is around $3817 and $3817 dollars per month, 68% possible!

My suggestion: if you want to invest your capital into this project, make sure that you have more than 15-20 servers, since this will keep you from losing money. 15-20 servers are also the best number of servers you will need to rent, because the total expected number of missions your servers will receive is just 43200 per day. If you hold more than 20 servers, some of your servers will not be used efficiently or cannot be used at all.

2.0 A few words

Thanks for providing this course. This is the best and most challenging course for me. However, I am enjoying it very much.

Thank you, Professor DiPierro.

Thank you, the grader.

Best Regards!

3.0 Complete Code of the project

```
import random
class webservice():
    time=30*24*60
    #assume one month, and 30 days per month
    server = []
    # how many servers
    cost = 0.0
    mission_assigned = 0.0
    queue_mission =[]
    # contains the mission on the queue.
    queue_time = []
    # contains the waiting time of each mission
    #a pareto PRNG
    def pareto(self,xm=5,alpha=2.5):
        return float(xm)*(1.-random.random())**(-1.0/alpha)

    def put_on_queue(self,x): # def a method to put next mission on queue
        for i in range(len(self.queue_mission)):
            if self.queue_mission[i] == 0:
                self.queue_mission[i] = x
                # return the index of queue where the mission should be placed.
                self.queue_time[i]= .0000001
                break

    def timer_queue(self,t):
```

```
# def a method that calculate the waiting time of the missions on queue
    for i in range(len(self.queue_time)):
        if self.queue_time[i] >0:
            self.queue_time[i] = self.queue_time[i]+t


def put_on_server(self):
    def a method put the first mission on queue onto server.
    for i in range(len(self.server)):
        if self.server[i] == 0.0:
            if self.queue_mission[0]>0:
                self.server[i] = self.queue_mission[0]
                # put the mission on server.
                self.queue_time.remove(self.queue_time[0])
                # First in First out
                self.queue_time.append(0.00000)
                self.queue_mission.remove(self.queue_mission[0])
                self.queue_mission.append(0.00000)


def timer_server(self,t):
    # def a method to manage the time
    for i in range(len(self.server)):
        self.server[i] = max(self.server[i]-t,0)


def drop_mission(self):
    # def a method to check whether the waiting time of the mission has come up to 5 minutes.
    cost = 0.0
```

```
    for i in range(len(self.queue_time)):
        #to shift the behind mission to fron spots to replace the one(s) that have past 5 minutes
        if self.queue_time[i] > 5:
            cost = cost + 10
            self.queue_time.remove(self.queue_time[i])
            self.queue_time.append(0.0000)
            self.queue_mission.remove(self.queue_mission[i])
            self.queue_mission.append(0.0000)
    return cost


def simulate_once(self,N):
    # N = # of servers, simulate a month.
    self.cost = N * 25
    # 25 is from 300/12, which is monthly cost for each server
    self.queue_time = [.0]*100
    self.queue_mission = [.0]*100
    self.server = [0.0 for i in range(N)]
    # how many servers
    self.mission_assigned = 0.0
    t = 0.0
    p =0.0
    m =0.0
    for i in range(self.time):
        m = random.expovariate(1.0)
        t = t + m
        self.mission_assigned += m
```

```
            while t>1:
                self.put_on_queue(self.pareto())
                t = t - 1
            p = self.drop_mission()
            while p>0:
                self.cost = self.cost+p
                p = self.drop_mission()
            self.put_on_server()
            self.timer_server(1)
            self.timer_queue(1)
        return self.cost
        ##to compute average cost per server, use self.cost/N instead.


    def profit(self,N,price):
    # def a method to get the profit, N is number of servers, price is the price for each mission finished.
        self.simulate_once(N)
        profit =(self.mission_assigned-(self.cost-25*N)/10)*price - self.cost
        return profit
        ##print "when price is",price,",","number of servers is",N,",", "profit is",profit


x = webservice()
list = []
for i in range(100): # create a list contains profit for 20 servers and price of 0.01
    list.append(x.profit(20,0.1))


def resample(list,n):
```

```
    # random resample for the list created and return a subset of the list
    return [random.choice(list) for i in range(n)]


def bootstrap(list,nboot=100):
    #bootstrap algorithm which returns sigma-mu and sigma+mu
    mu = sum(list)/len(list)
    mu_samples = [ ]
    for i in range(nboot):
        sample = resample(list,len(list))
        mu_sample = sum(sample)/len(sample)
        mu_samples.append(mu_sample)
    mu_samples.sort()
    print "sigma-mu=",mu_samples[15], "mu=",mu ,"sigma+mu=",mu_samples[100-16]
```