CS 237: Homework 12: Discrete-Event Simulation of Queueing System

Due date: PDF file due Wednesday December 11th@ 11:59PM in GradeScope

Homeworks may be handed in late up to noon Sunday 12/15 with no penalty. There will be possible questions on the final exam about this homework, as well as the previous one.

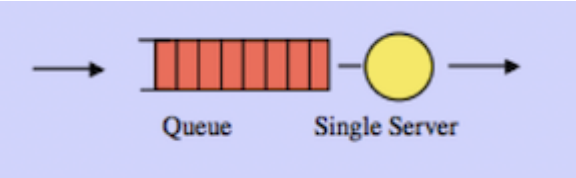
Please complete this notebook by filling in solutions where indicated. Be sure to "Run All" from the Cell menu before submitting.

You may use ordinary ASCII text to write your solutions, or (preferably) Latex. A nice introduction to Latex in Jupyter notebooks may be found here: http://data-blog.udacity.com/posts/2016/10/latex-primer/ (http://data-blog.udacity.com/posts/2016/10/latex-primer/)

As with previous homeworks, just upload a PDF file of this notebook. Instructions for converting to PDF may be found on the class web page right under the link for homework 1.

General Instructions

In this homework we will simulate a simple system consisting of a single queue and a single server:



Although this is a very broad idea, we will focus on simulating a ready queue which holds tasks waiting for the CPU.

How should we approach this simulation? Previously, we did almost all of our simulations using discrete time steps:

```
        for k in range(num_trials):     # you can think of this as one trial e
very time step k
            ....
```

However, it is very unrealistic to model queues using such a technique, since most queues exist in an environment when tasks are independent, asynchronous, and arrive at a random time which is best modeled as a real number, not an integer. We will assume a Poisson arrival process, so we can model the inter-arrival time using an exponential distribution.

It is also the case that the service time (the amount of time the task will take in the CPU) will be modeled as a real number, and in fact, experiments show that the exponential distribution is also a good model for the service times.

But now, because we can no longer use a `for` loop to model time in our simulation, we will have to use a very different approach, which we describe in the rest of this section.

Discrete-Event Simulation

As an overview, I can not do better than the beginning of the Wikipedia article:

> "In the field of simulation, a discrete-event simulation (DES) models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next. This contrasts with continuous simulation in which the simulation continuously tracks the system dynamics over time. Instead of being event-based, this is called an activity-based simulation; time is broken up into small time slices and the system state is updated according to the set of activities happening in the time slice. Because discrete-event simulations do not have to simulate every time slice, they can typically run much faster than the corresponding continuous simulation. Another alternative to event-based simulation is process-based simulation. In this approach, each activity in a system corresponds to a separate process, where a process is typically simulated by a thread in the simulation program. In this case, the discrete events, which are generated by threads, would cause other threads to sleep, wake, and update the system state."

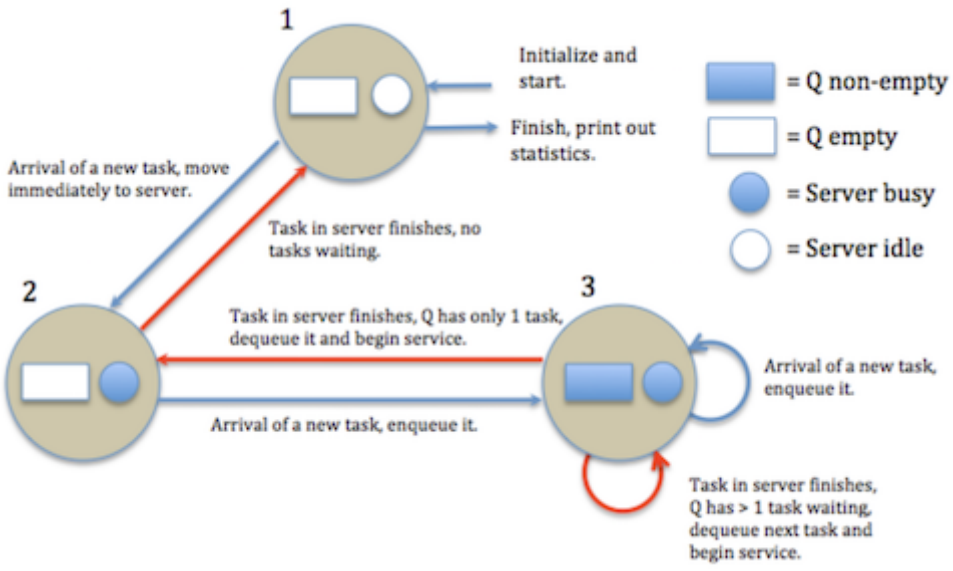
The essential components of a DES are the following:

Clock: A variable keeping track of the current time in seconds (a float). This most important thing to understand about the clock is that it is not like the counter in a for loop, counting off the seconds; rather, the main loop performs one event per iteration, and the clock keeps track of the time elapsed, skipping ahead from one event time stamp to the next. In general, the time interval between events will not be constant.

Events List: This is an ordered list of events, each with a code telling what is the kind of action to take, and a time stamp for when the event is to occur, and any other information needed for the event. At each iteration of the main loop, the next event will be removed and the action taken. The usual data structure for this list is a priority queue implemented as a min-heap.

State Transitions in the DES

The simulation will go through various changes in configuration or **state transitions** during execution. In some DES's, the number of states is so large, you would literally write the code with a big loop to determine what state the simulation will go to next. For us, there are really only 3 states, and the simulation code doesn't strictly adhere to this diagram; however, it may be useful for understanding what is happening:



Data Structures for the DES

An **event** will be represented by a triple

```
(event_time, event_type, task)
```

The event_type is an integer code with the following meanings:

- 0 = arrival event, a task arrives in the queue: (0, arrival_time, task)
- 1 = finish event, a task

Data Structures for the DES

An **event** will be represented by a triple

```
(event_time, event_type, task)
```

The event_type is an integer code with the following meanings:

- 0 = arrival event, a task arrives in the queue: (0, arrival_time,)
- 1 = finish event, a task finishes service and leaves the CPU (1, finish time,)

Tasks are represented by a list

```
[ arrival_time, service_time, queue_wait_time ],
```

The queue wait time will be initialized to 0, and updated when a task leaves the queue and starts service.

The **events list** will be implemented by a heap, using the Python heapq library (see [this (https://docs.python.org/2/library/heapq.html)](https://docs.python.org/2/library/heapq.html) link for details, including examples of how to use it on tuples); the two functions used are:

```
   heapq.heappush(heap, item)  #Push the value item onto the heap (just a li
st), maintaining the heap ordering.

   heapq.heappop(heap)   # Pop and return the smallest item from the heap, m
aintaining the heap ordering.
                         # If the heap is empty, IndexError is raised. To ac
cess the smallest
                         # item without popping it, use heap[0].
```

The **Task Queue** will also use a heap, ordered in two different ways:

- FIFO (First-In-First-Out, "First come first served"): The default and simplest implementation; the heap is ordered by arrival time;
- SJF (Shortest-Job-First): An implementation which often increases throughput; the heap is ordered by service time, shortest time first.

Other details of the simulation:

- An initial list of tasks will be generated before the simulation begins, and the simulation will run until all tasks have been processed.
- Arrival of tasks will follow a Poisson distribution with rate parameter `lam` = arrivals per unit time, but expressed by an exponential `Exp( lam )` which will give the inter-arrival time; by successively adding the inter-arrival times we arrive at the arrival times (which increase throughout the simulation).
- The service time of tasks follows a **exponential distribution** `Exp( beta )`, where `beta = 1 / (mean service time)`; we can think of `beta` as the rate at which tasks need the CPU or the rate at which tasks finish and leave the system;
- In order to simulate various kinds of loads on the system, it is only necessary to investigate the relationship between `lam` and `beta`, in particular when:

  - `lam` $<$ `beta` : Tasks finish and depart at a faster rate then they arrive; such a system is *under-loaded*;
  - `lam` $=$ `beta` : Arrival and departure *mean rates* are equally matched; and
  - `lam` $>$ `beta` : Tasks arrive faster than they can be served; such a system is *over-loaded*.

- We will be very interested in watching what happens to the system as we change this relationship, as the

A pseudo-code version of the algorithm used to run the simulation is as follows:

```
num_tasks = 10**4

current_time = 0

mean_arrival_rate = 100

task_list = []

queue = []

Create an array of num_tasks arrival tasks with exponentially-distributed in
ter-arrival times and service times;
     the mean of the service times will be varied (see below).
     Note that the interarrival time need to be added to the last arrival ti
me, to create an increasing sequence
     of time stamps for arrival events.

while(events list is non-empty) {
    e = get_next_event()
    clock = time stamp of e
    if e is an arrival event
        insert the task into the queue, following the queue discipline being
used (FIFO or SJF)
    else # must be a finish event
         if the queue is non-empty
             remove the task at the head of the queue and create a finish ev
ent for that task,
                with time stamp and queue wait time as shown in boldface her
e:

                    ( (current_time + service_time), 1, [ arrival_time, serv
ice_ time, (current_time - arrival_time) ] )



Print out the statistics.
```

In [1]:
```
    # Imports potentially used for this lab


import matplotlib.pyplot as plt   # normal plotting

import math
from random import seed, random, uniform, randint
import numpy as np
import scipy.signal           # used to smooth graphs

from collections import Counter

%matplotlib inline

# Calculating permutations and combinations efficiently

def P(N,K):
    res = 1
    for i in range(K):
        res *= N
        N = N - 1
    return res

def C(N,K):
    if(K < N/2):
        K = N-K
    X = [1]*(K+1)
    for row in range(1,N-K+1):
        X[row] *= 2
        for col in range(row+1,K+1):
            X[col] = X[col]+X[col-1]
    return X[K]

# Useful code from lab 01

# This function takes a list of outcomes and a list of probabilities and
# draws a chart of the probability distribution.

def draw_distribution(Rx, fx, title='Probability Distribution for X'):
    plt.figure(figsize=(8, 4))
    plt.bar(Rx,fx,width=1.0,edgecolor='black')
    plt.ylabel("Probability")
    plt.xlabel("Outcomes")
    if (Rx[-1] - Rx[0] < 30):
        ticks = range(Rx[0],Rx[-1]+1)
        plt.xticks(ticks, ticks)
    plt.title(title)
    plt.show()

# This function takes a list of outcomes, calculates a histogram, and
# then draws the empirical frequency distribution.

def show_distribution(outcomes, title='Empirical Probability Distributio
n'):
    num_trials = len(outcomes)
    Rx = range( int(min(outcomes)), int(max(outcomes))+1 )
```

```
    freqs = Counter(outcomes)
    fx = [freqs[i]/num_trials for i in Rx]
    draw_distribution(Rx, fx, title=title)

def round4(x):
    return round(x+0.00000000001,4)

def round4_list(L):
    return [ round4(x) for x in L]
```

In [2]:
```
# Two kinds of queues

import heapq

arrival_event = 0
finish_event = 1

def task_to_string(task):
    return str([task[0], round4(task[1]),round4(task[2])])

class FIFO_Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)

    def show(self):
        if self.isEmpty():
            return "empty\n"
        ret = ""
        for k in range(len(self.items)-1,-1,-1):
            ret += "\n\t" + task_to_string(self.items[k])
        return  ret+"\n"

class SJF_Queue:
    def __init__(self):
        self.heap = []

    def isEmpty(self):
        return self.heap == []

    def enqueue(self, item):
        # wrap the task in a tuple so can be heap ordered
        tup = (item[1],item)
        heapq.heappush(self.heap, tup)

    def dequeue(self):
        return heapq.heappop(self.heap)[1]

    def size(self):
        return len(self.heap)

    def show(self):
        if self.isEmpty():
            return "empty\n"
        ret = ""
```

```
        for k in range(len(self.heap)):
            ret += "\n\t" + task_to_string(self.heap[k][1])
        return ret+"\n"


def event_to_string(event):
    if event[1] == arrival_event:
        return "( " + str(round4(event[0])) + ", arrival, " + task_to_st
ring(event[2]) + " )"
    else:
        return "( " + str(round4(event[0])) + ", finish, " + task_to_str
ing(event[2]) + " )"

class Event_Queue:
    def __init__(self):
        self.heap = []

    def isEmpty(self):
        return self.heap == []

    def enqueue(self, item):
        heapq.heappush(self.heap, item)

    def dequeue(self):
        return heapq.heappop(self.heap)

    def size(self):
        return len(self.heap)

    def show(self):
        if self.isEmpty():
            return "empty\n"
        ret = ""
        for k in range(len(self.heap)):
            ret += "\n\t" + event_to_string(self.heap[k])
        return ret+"\n"
```

In [3]:
```
# Testing the queues

print("\nNote: printouts of queues and heaps are with head or min value
 at top;")
print("remember that heaps are not in decreasing order but heap ordered.
\n")

print("FIFO")
Q = FIFO_Queue()
Q.enqueue([0,.5,.4,0])
Q.enqueue([1,.1,.2,0])
Q.enqueue([2,.2,.3,0])

print(Q.size())
print(Q.isEmpty())
print(Q.show())
print(Q.dequeue())
print(Q.dequeue())
print(Q.size())
print(Q.isEmpty())
print()

print("SJF")
Q2 = SJF_Queue()
Q2.enqueue([0,.5,.4,0])
Q2.enqueue([1,.1,.2,0])
Q2.enqueue([2,.2,.3,0])
print(Q2.size())
print(Q2.isEmpty())
print(Q2.show())
print(Q2.dequeue())
print(Q2.dequeue())
print(Q2.size())
print(Q2.isEmpty())
print()

print("Event")
H = Event_Queue()
H.enqueue((61,0,[0,61,3,4]))
H.enqueue((9,1,[1,9,3,4]))
H.enqueue((1,0,[2,1,3,4]))
H.enqueue((19,0,[3,19,3,4]))
H.enqueue((0,0,[4,0,3,4]))
H.enqueue((99,0,[5,99,3,4]))
print(H.size())
print(H.isEmpty())
print(H.show())
while( not H.isEmpty()):
    print(event_to_string(H.dequeue()))

print(H.size())
print(H.isEmpty())
print(H.show())
```

```
Note: printouts of queues and heaps are with head or min value at top;
remember that heaps are not in decreasing order but heap ordered.

FIFO
3
False

        [0, 0.5, 0.4]
        [1, 0.1, 0.2]
        [2, 0.2, 0.3]

[0, 0.5, 0.4, 0]
[1, 0.1, 0.2, 0]
1
False

SJF
3
False

        [1, 0.1, 0.2]
        [0, 0.5, 0.4]
        [2, 0.2, 0.3]

[1, 0.1, 0.2, 0]
[2, 0.2, 0.3, 0]
1
False

Event
6
False

        ( 0.0, arrival, [4, 0.0, 3.0] )
        ( 1.0, arrival, [2, 1.0, 3.0] )
        ( 9.0, finish, [1, 9.0, 3.0] )
        ( 61.0, arrival, [0, 61.0, 3.0] )
        ( 19.0, arrival, [3, 19.0, 3.0] )
        ( 99.0, arrival, [5, 99.0, 3.0] )

( 0.0, arrival, [4, 0.0, 3.0] )
( 1.0, arrival, [2, 1.0, 3.0] )
( 9.0, finish, [1, 9.0, 3.0] )
( 19.0, arrival, [3, 19.0, 3.0] )
( 61.0, arrival, [0, 61.0, 3.0] )
( 99.0, arrival, [5, 99.0, 3.0] )
0
True
empty
```

In [4]:
```
# codes for event types

arrival_event = 0
finish_event = 1

def run(tasks, queue_discipline, trace=False):

    clock = 0    # current time in the simulation

    if queue_discipline == "FIFO":
        task_queue = FIFO_Queue()
    else:
        #print("using SJF")
        task_queue = SJF_Queue()

    event_list = Event_Queue()

    for k in range(len(tasks)):
        event_list.enqueue( (tasks[k][0],arrival_event,tasks[k] ) )

    finished_tasks = FIFO_Queue()

    CPU = []        # empty list indicates CPU is idle

    if trace:
        print("\nExecution Trace")
        print("\nNote: printouts of queues and heaps are with head or mi
n value at top;")
        print("remember that heaps are not in decreasing order but heap
 ordered.\n")

    while(not (event_list.isEmpty())):

        if trace:
            print("\n==================================================
==========")
            print("\nClock: " + str(round4(clock)))
            print("\nEvents: " + event_list.show())
            print("Task queue: " + str(task_queue.show()))
            if CPU != []:
                print("CPU processing task " + task_to_string(CPU))
            else:
                print("CPU idle")

            print("\nFinished tasks: " + str(finished_tasks.show()))

        # get the next event
        next_event = event_list.dequeue()

        if trace:
            print("----------------------------------------------------
---------")

        if trace:
            print("\nNext Event: \n\t" + event_to_string(next_event))
```

```
        # update the clock to this event's time
        clock = next_event[0]

        # arrival event
        if next_event[1] == arrival_event:
            if CPU == []:                # Q and CPU both empty, no wai
t, insert finish event
                if trace:
                    print("\nCPU and task queue both empty, task goes di
rectly to CPU, create finish event.")
                task = next_event[2]
                arrival_time = task[0]
                service_time = task[1]
                task[2] = clock - arrival_time           # queue_wait_t
ime
                finish_time = clock + service_time
                event_list.enqueue(  ( finish_time, finish_event, task )
)
                CPU = task
            else:
                task_queue.enqueue(next_event[2])
                if trace:
                    print("\nCPU busy, insert arriving task in task queu
e.")

        # finish event, so put task in list of finished tasks and get ne
xt task from queue if exists
        else:
            finished_tasks.enqueue(next_event[2])
            if trace:
                print("\nTask in CPU finishes.")
            CPU = []
            if not task_queue.isEmpty():
                if trace:
                    print("\nDequeue next task and start to run in CPU,
 create finish event.")
                task = task_queue.dequeue()
                arrival_time = task[0]
                service_time = task[1]
                task[2] = clock - arrival_time           # queue_wait_t
ime
                finish_time = clock + service_time
                event_list.enqueue(  ( finish_time, finish_event, task )
)
                CPU = task
            elif trace:
                    print("\nTask queue empty, CPU becomes idle.")
    if trace:
        print("\nAll tasks complete, simulation ends.")
        print("\nFinished tasks: " + str(finished_tasks.show()))
        print("=============================================================
=====\n")

    tasks = []
    while not finished_tasks.isEmpty():
        tasks.append(finished_tasks.dequeue())
```

```
    return tasks
```

In [5]:
```
# this prints out GANNT charts and charts for CPU
# utilization, queue length, and queue length distribution.

def analyzeResults(task,showCharts=True,showStats=True):

    # Determine various parameters and means

    numTasks = len(task)

    totalTime = task[-1][0]+task[-1][1]+task[-1][2]

    meanServiceTime = np.mean([task[k][1] for k in range(len(task))])

    meanInterarrivalTime =  task[-1][0] / len(task)

    meanWaitTime = np.mean([task[k][2] for k in range(len(task))])

    cpuUtilization = sum( [ task[k][1] for k in range(len(task))]) / tot
alTime

    increment = 0.001                        # use to create charts and c
ollect stats

    X = np.arange(0,totalTime,increment)

    if showCharts:

        # Print GANTT Chart

        fig = plt.figure(figsize=(15,10))
        fig.subplots_adjust(hspace=.5)
        ax1 = fig.add_subplot(311)
        #plt.yticks(range(len(task)))
        plt.ylim((-0.5,len(task)))
        plt.title('GANNT Chart')
        plt.ylabel('Task Number')
        plt.xlabel('Time (sec)')

        if len(task) > 50:
            lw = 1
        elif len(task) > 20:
            lw = 2
        elif len(task) > 10:
            lw = 3
        else:
            lw = 4

        for k in range(len(task)):
            arrival = task[k][0]
            begin_service = arrival + task[k][2]
            end_service = begin_service + task[k][1]
            plt.hlines(k, arrival, begin_service, color='C0',linestyle=
'dotted',linewidth=lw)
            plt.hlines(k, begin_service, end_service, color='C0',linesty
le='solid',linewidth=lw)
```

```
    # Print CPU Utilization Chart

    Y = np.zeros(len(X))

    for k in range(len(task)):
        arrival = task[k][0]
        begin_service = arrival + task[k][2]
        end_service = begin_service + task[k][1]
        for x in np.arange(begin_service,end_service,increment):
            Y[int(x/increment)] = 1

    Y = scipy.signal.medfilt(Y)
    Y[-1] = 0

    fig.add_subplot(312,sharex=ax1)
    #plt.yticks(range(len(task)))
    plt.title('CPU Utilization')
    plt.ylabel('Usage')
    plt.xlabel('Time (sec)')
    plt.ylim((-0.15,1.2))
    plt.plot(X,Y)

# Plot Queue length over time

Y1 = np.zeros(len(X))

for k in range(len(task)):
    arrival = task[k][0]
    begin_service = arrival + task[k][2]
    for x in np.arange(arrival,begin_service,increment):
        Y1[int(x/increment)] += 1

Y1 = scipy.signal.medfilt(Y1)

Y1[-1] = 0

if showCharts:

    maxLength = int(max(Y1))

    fig.add_subplot(313,sharex=ax1)
    #plt.yticks(range(maxLength+1))
    plt.title('Queue Length Over Time')
    plt.ylabel('Queue Length')
    plt.xlabel('Time (sec)')
    plt.ylim((-0.5,maxLength+ 0.5))
    plt.plot(X,Y1)
    plt.show()

    show_distribution(Y1,title="Distribution of Queue Lengths")


maxQueueLength = max(Y1)
meanQueueLength = np.mean(Y1)
stdQueueLength = np.std(Y1)
```

```
    if showStats:
        print("\nNumber of Tasks:\t" + str(numTasks))
        print("Total Time:\t\t" + str(round4(totalTime)))
        print("Mean Interarrival Time:\t" + str(round4(meanInterarrivalT
ime)))
        print("Mean Wait Time:\t\t" + str(round4(meanWaitTime)))
        print("Mean Service Time:\t" + str(round4(meanServiceTime)))
        print("CPU Utilization:\t" + str(round4(cpuUtilization)))
        print("Maximum queue length:\t" + str(round4(maxQueueLength)))
        print("Mean queue length:\t" + str(round4(meanQueueLength)))
        print("Std of queue length:\t" + str(round4(stdQueueLength)))

    return [numTasks,totalTime,meanInterarrivalTime,meanWaitTime,
            meanServiceTime,cpuUtilization,
            maxQueueLength,meanQueueLength,stdQueueLength]
```

Problem Zero

Nothing to hand in here, but you need to understand how tasks are represented, and how they proceed through the system before you can do the probability problems to follow.

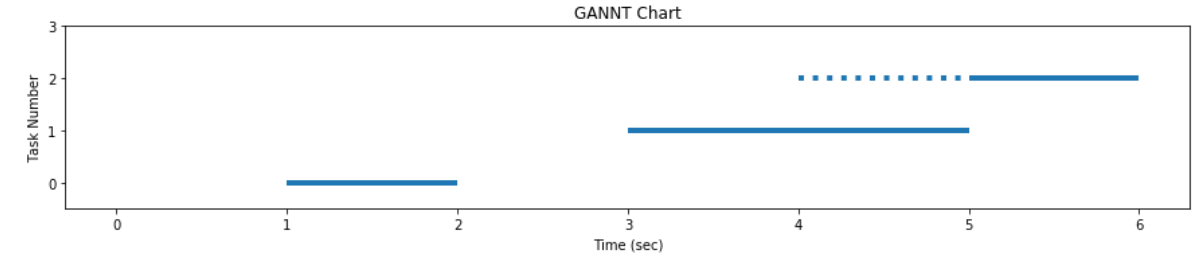
Each **task** is represented by a list

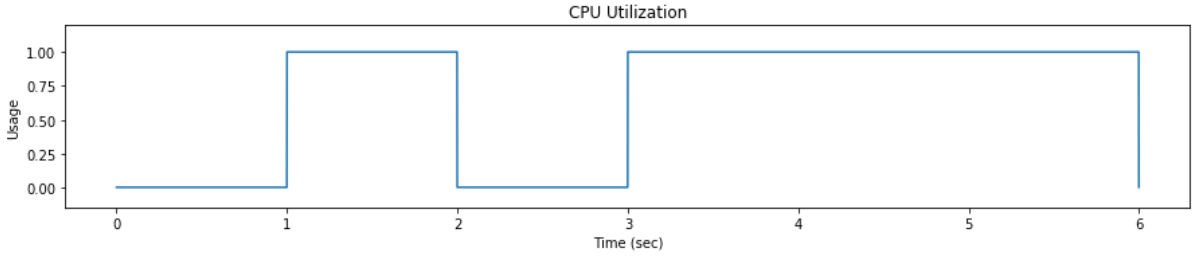
```
[ arrival_time, service_time, queue_wait_time ],
```

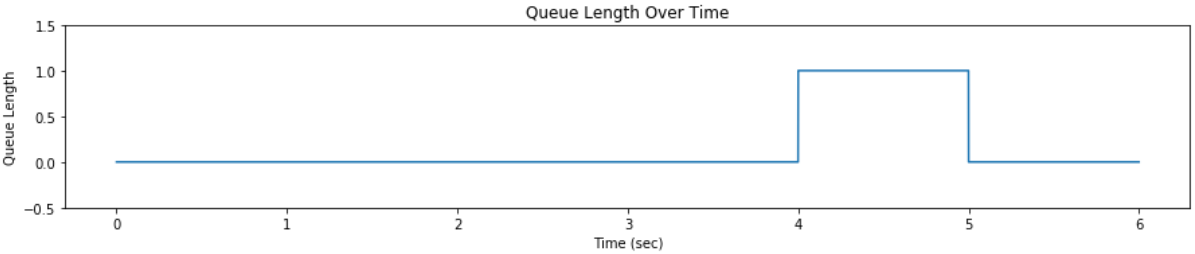
where the list of tasks is given in order of arrival time, and the last parameter is 0 (it will be filled in by the simulation and the resulting list returned by the simulation).

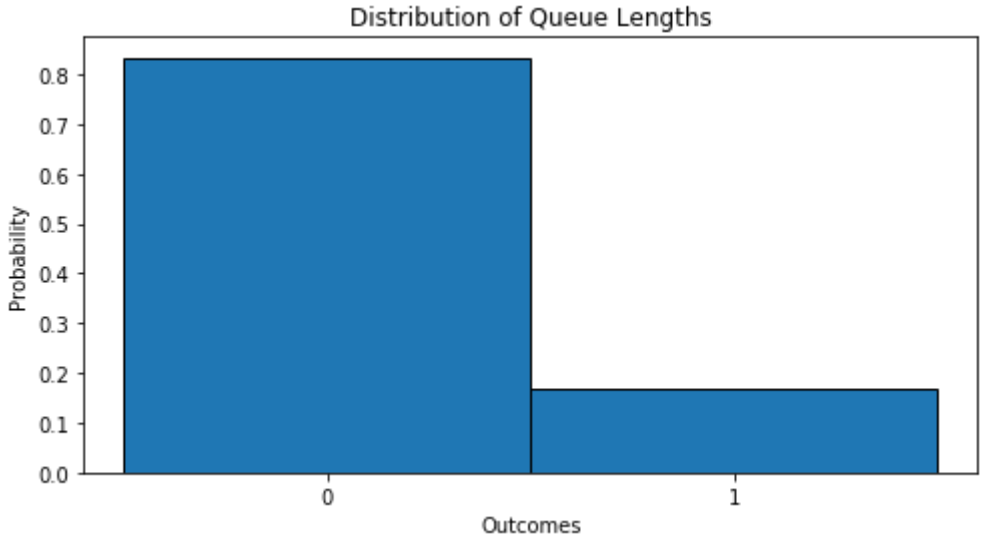
Play around with these examples, including changing the parameters to try different queue disciplines (FIFO or Shortest Job First), and exhaustive tracing of the simulation. When you are comfortable with the framework, continue with the rest of the problems.

In [6]:
```
tasklist1 = [[1, 1, 0],           # Basic example
             [3, 2, 0],           # FIFO vs SJF will not make a differen
ce here
             [4, 1, 0]]

tasklist2 = [[1, 4, 0],           # This example will show the differenc
e between FIFO and SJF
             [2, 5, 0],           # In FIFO, tasks executed as they arri
ve (tasks 0,1,2,3)
             [3, 2, 0],           # In SJF, tasks will be executed in or
der
             [6, 3, 0]]

tasklist3 = [[0.1, 0.1, 0],       # A more complicated example!
             [0.3, 0.4, 0],       # Will also behave differently under F
IFO and SJF
             [0.5, 0.2, 0],
             [0.6, 0.3, 0],
             [0.8, 0.2, 0]]

tasklist4 = [[0,2,0], [1.1,3,0], [3.4,4,0], [3.45,3,0], [5,3.23,0],[6.2,
2.9,0],[6.4,2.32,0], [9.99,1.2,0], [10.3,3.4,0], [12.8,3.9,0], [15.2,3.4
,0],
       [15.67,2.43,0],[17.01,2.8,0],[18.8,2.2,0],[20.1,2.99,0],[21.7,5.3
4,0],[24.4,2.2,0]]

tasks = tasklist1
#tasks = tasklist2
#tasks = tasklist3
#tasks = tasklist4

# run the simulation with the given task list
# the second parameters gives the queue discipline, either "FIFO" or "SJ
F"
# the third parameter controls whether an exhaustive trace is done of th
e simulation

finished = run(tasks,"FIFO",False)
#finished = run(tasks,"SJF",False)          # use shortest job first sc
heduling
#finished = run(tasks,"FIFO",True)          # exhaustive tracing
#finished = run(tasks,"SJF",True)

#print(finished)

# Analyze the results, printing out various charts, and return relevant
 statistics

stats = analyzeResults(finished)

# stats = [numTasks,totalTime,meanInterarrivalTime,meanWaitTime,meanServ
iceTime,
#          cpuUtilization,maxQueueLength,meanQueueLength,stdQueueLength]
```





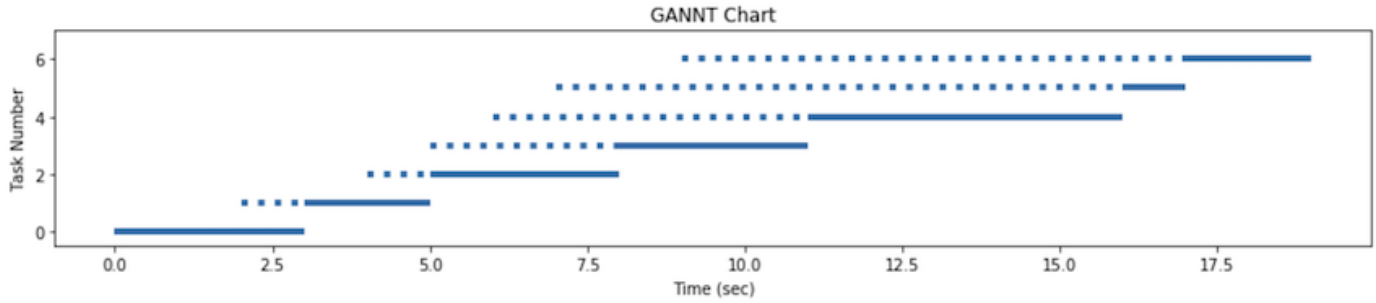




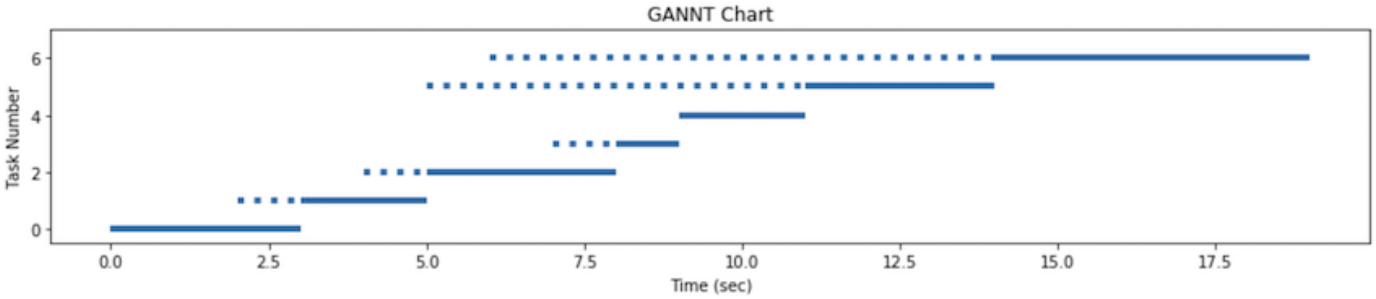

```
Number of Tasks:         3
Total Time:              6.0
Mean Interarrival Time: 1.3333
Mean Wait Time:          0.3333
Mean Service Time:       1.3333
CPU Utilization:         0.6667
Maximum queue length:    1.0
Mean queue length:       0.1667
Std of queue length:     0.3727
```

Problem One

(A) Give a task list `ans1` which will produce the following Gannt Chart using FIFO:

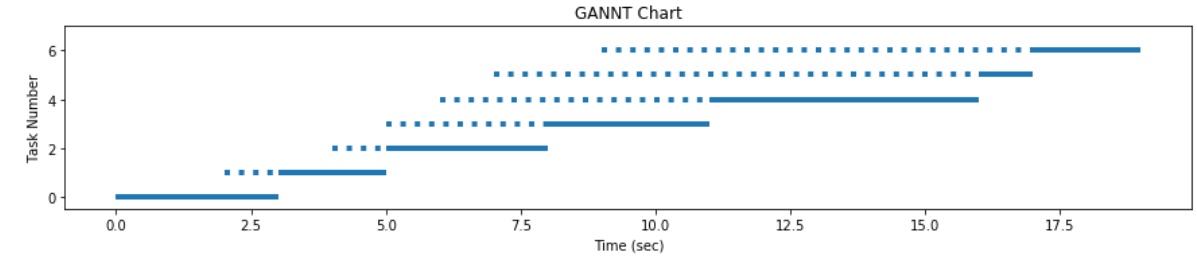


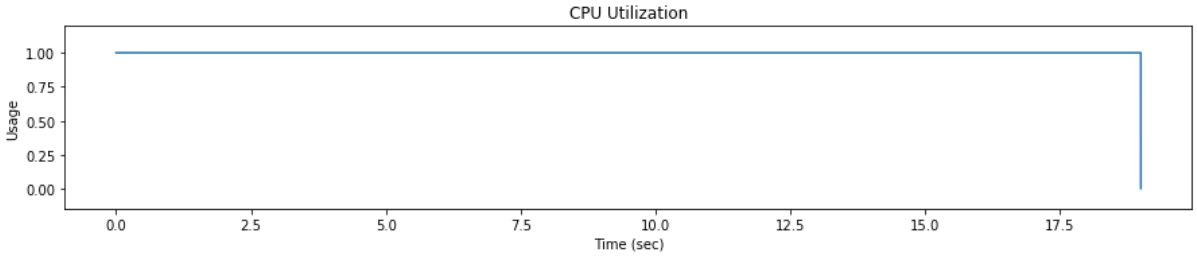
(B) Give a task list `ans2` which will produce the following Gannt Chart using SJF:

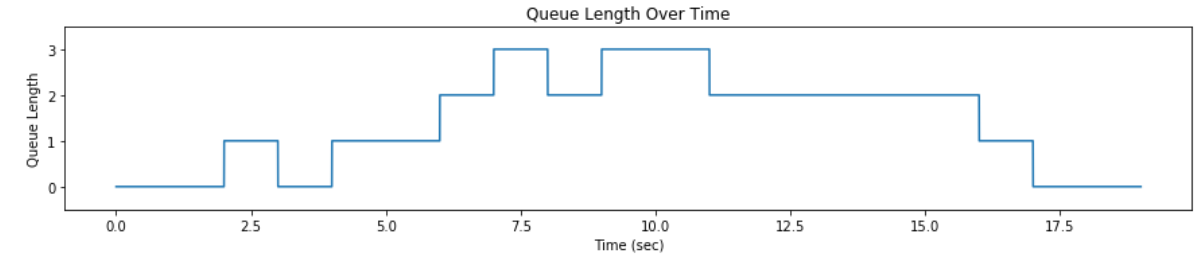


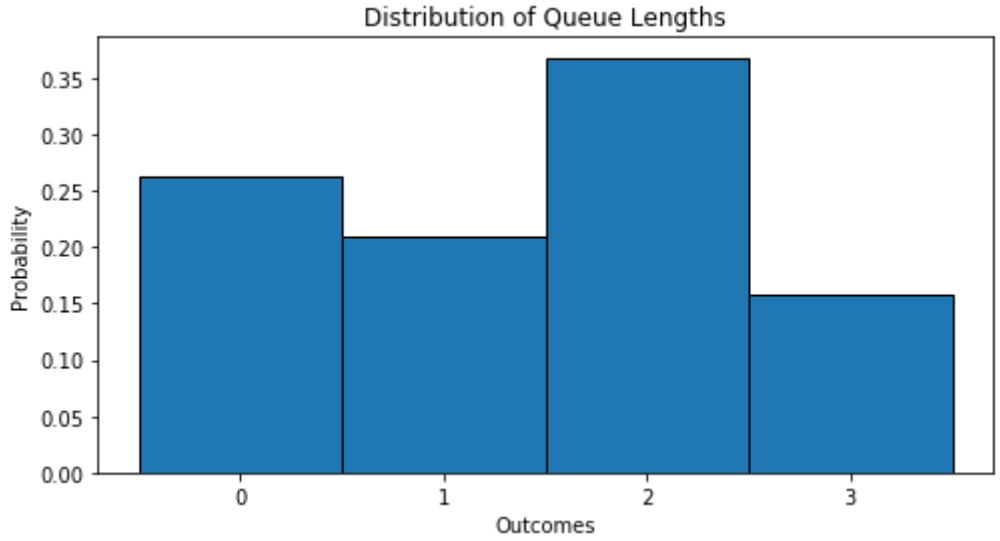
(Hint: All arrival times and service requests will be integers.)

In [20]:
```
ans1a = [ [0,3,0], [2,2,0], [4,3,0], [5,3,0], [6,5,0], [7,1,0], [9,2,0]
]    # your answer here

stats = analyzeResults(run(ans1a,"FIFO",False))

ans1b = [ [0,3,0], [2,2,0], [4,3,0], [5,3,0], [6,5,0], [7,1,0], [9,2,0]
]     # your answer here

stats = analyzeResults(run(ans1b,"SJF",False))
```

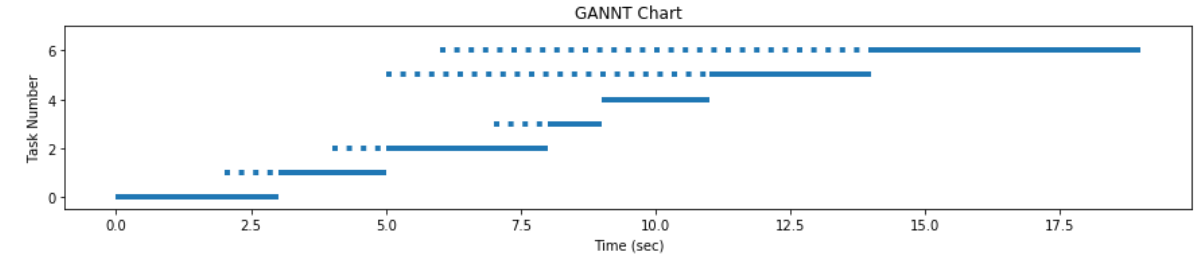


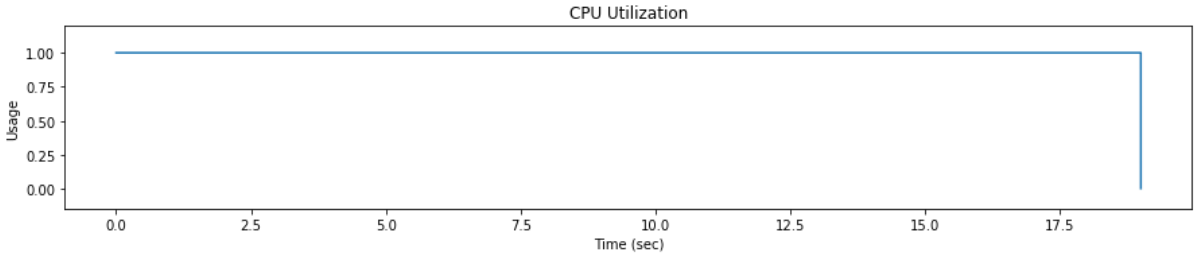


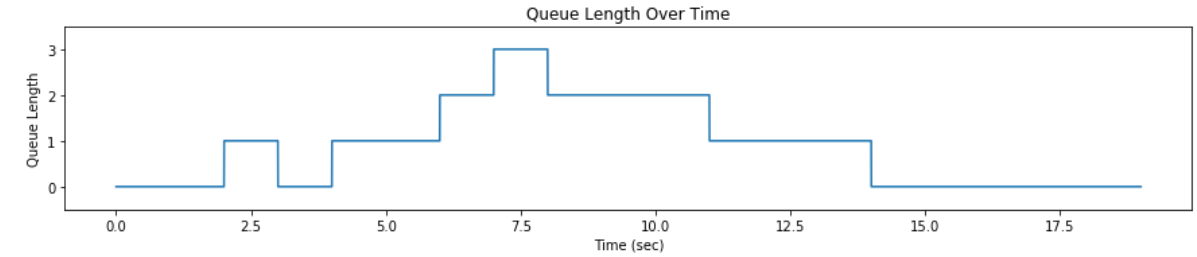


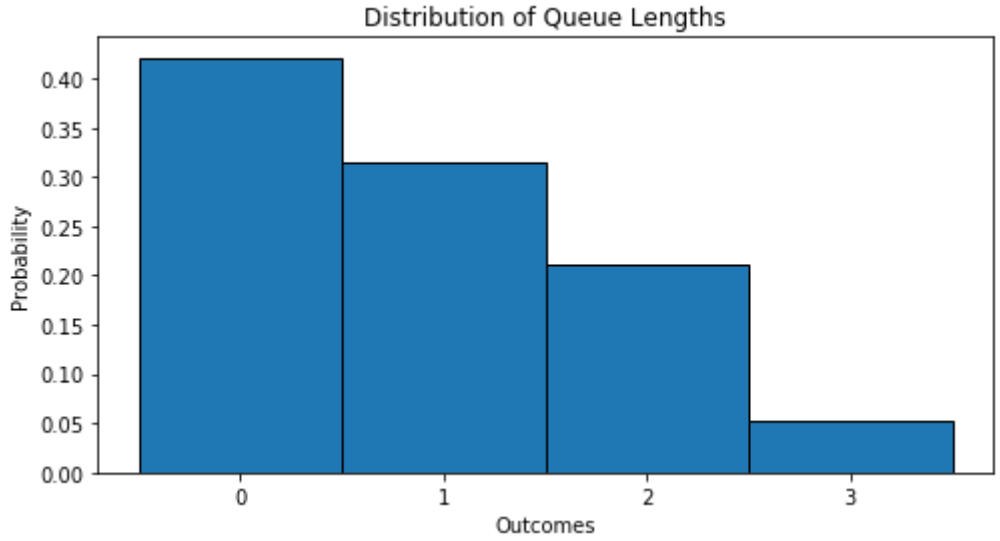



```
Number of Tasks:         7
Total Time:              19.0
Mean Interarrival Time: 1.2857
Mean Wait Time:          3.8571
Mean Service Time:       2.7143
CPU Utilization:         1.0
Maximum queue length:    3.0
Mean queue length:       1.4209
Std of queue length:     1.0421
```










```
Number of Tasks:         7
Total Time:              19.0
Mean Interarrival Time: 0.8571
Mean Wait Time:          2.4286
Mean Service Time:       2.7143
CPU Utilization:         1.0
Maximum queue length:    3.0
Mean queue length:       0.8947
Std of queue length:     0.9116
```

Problem Two: Generating a Task List

Now we are going to generate a list of tasks following an Exponential Distribution for both the interarrival time and the service time. This is a very common assumption when doing these kind of queueing simulations. As described above, we will let

- lam = the mean arrival rate of tasks in the system, and
- beta = 0.1 = mean service rate requested by tasks

but only `lam` will be changed during the simulation, from below 10 (underload) to above 10 (overload).

Recall that **tasks** are represented by a list

```
[ arrival_time, service_time, queue_wait_time ],
```

where the last component is initialized to 0, and set during the simulation. From the list of completed tasks, it is possible to generate all the statistics discussed above (this will already be done for you in the code).

We will use the `scipy.stats` function `expon.rvs(scale)` to generate the random variates for the interarrival times of tasks, and also for the service times of tasks.

This Python function uses scale = β (the mean of the distribution) as a parameter, so you have to be careful, since it is the exact opposite of the way these distributions are described in the literature:

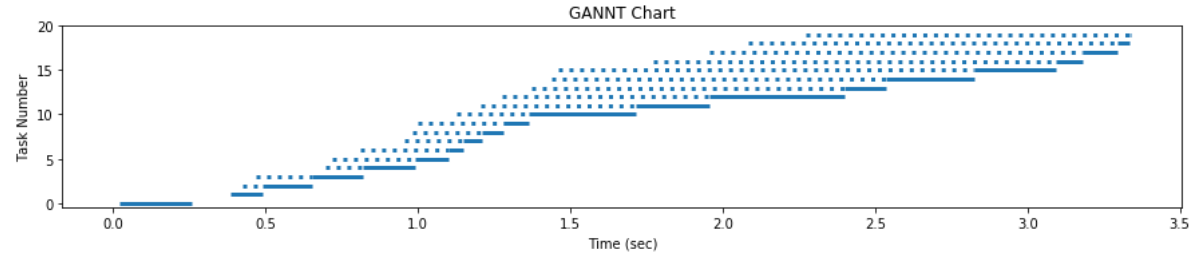
- If you are interested in generating exponential variates with mean `beta`, then you call `expon.rvs(scale=beta)`, but
- If you are interested in interarrival times of a Poisson process with arrival rate of `lam` arrivals per unit time, then you would call `expon.rvs(scale=1/lam)`.

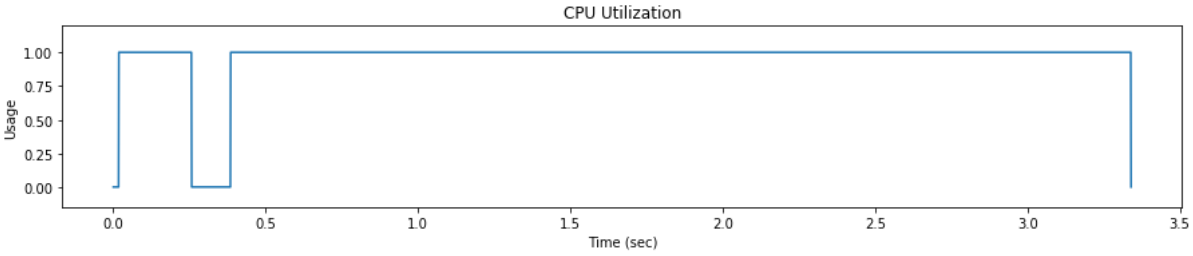
Part (A):

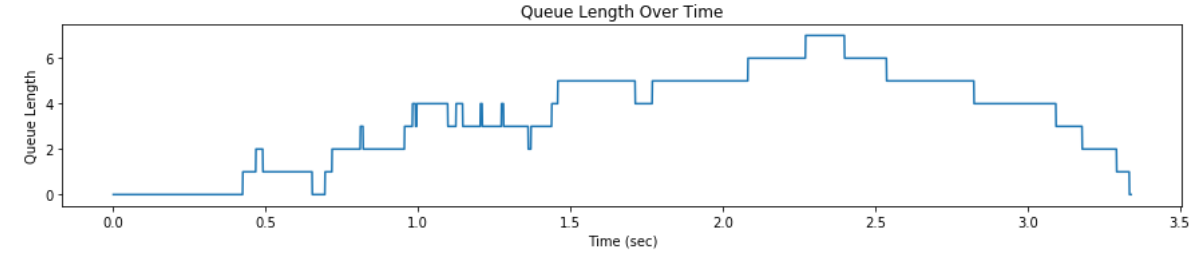
Complete the following to generate a list of tasks to input into the system, and then run the simulation and observe the results.

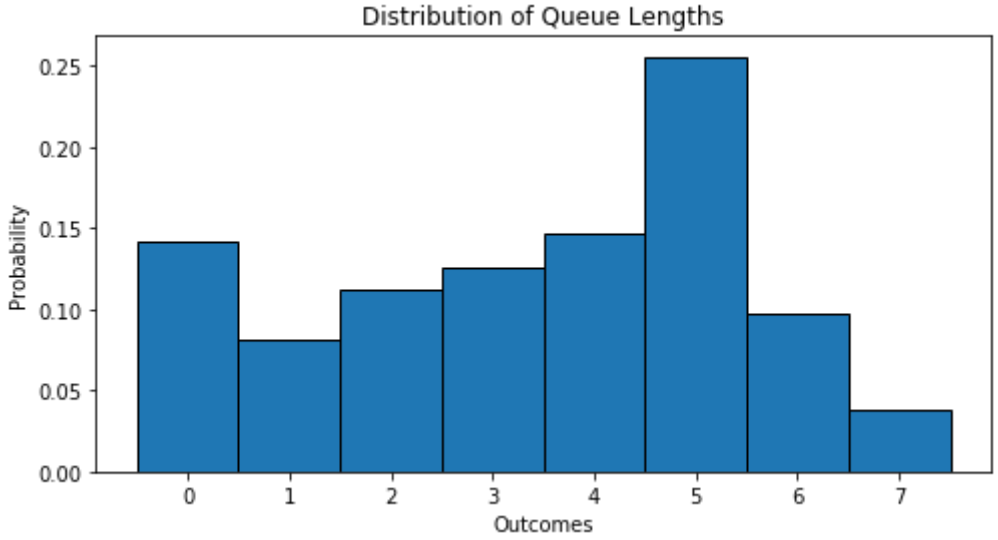
Hint: Don't forget that the inter-arrival times have to be added together to get the succession of actual arrival times.

In [8]:
```
from scipy.stats import expon

def getTaskList(num_tasks,lam,beta):
    time = 0
    tasks = []
    for k in range(num_tasks):
        time += expon.rvs(scale=1/lam)
        tasks.append( [time,expon.rvs(scale=beta),0] )
    return tasks

# parameters for the simulation

num_tasks = 20
lam = 10      # mean arrival rate of tasks in the system -- this is the
 only parameter we will change
beta = 0.1   # mean service time of arriving tasks -- do not change this
parameter

stats = analyzeResults(run(getTaskList(num_tasks,lam,beta),"FIFO",False
))
```







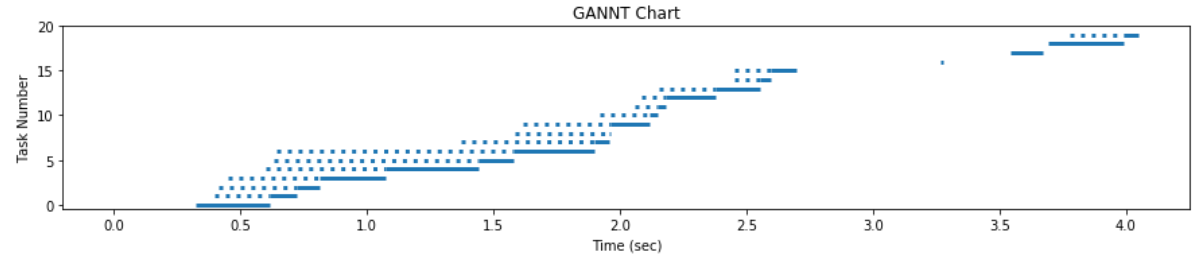


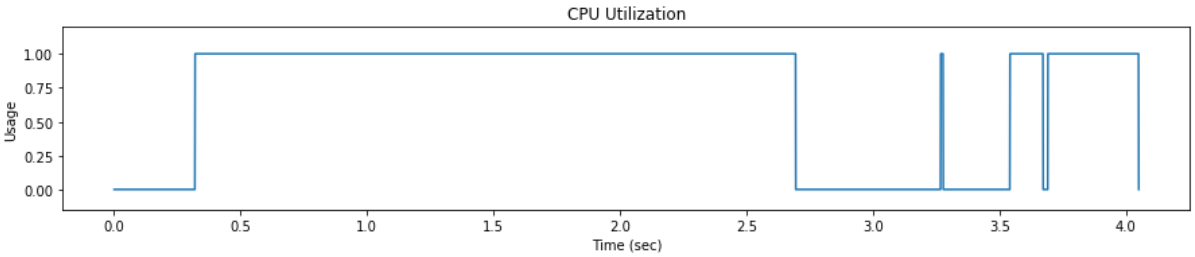

```
Number of Tasks:         20
Total Time:              3.3408
Mean Interarrival Time: 0.1136
Mean Wait Time:          0.5687
Mean Service Time:       0.1597
CPU Utilization:         0.956
Maximum queue length:    7.0
Mean queue length:       3.4077
Std of queue length:     2.0513
```

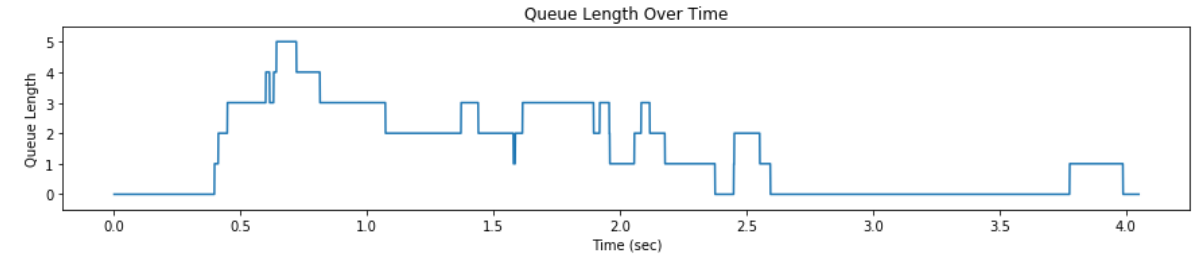
Part (B)

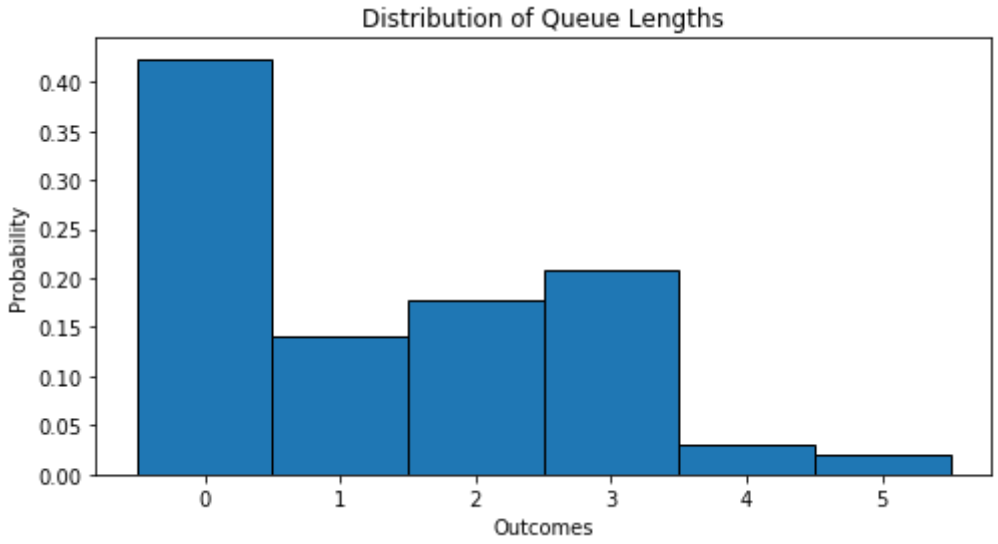
Run the same simulation, but with the arrival rate set much lower than 10, say at 5.

In [9]:
```
lam = 5
print(analyzeResults(run(getTaskList(num_tasks,lam,beta),"FIFO",False)))
```







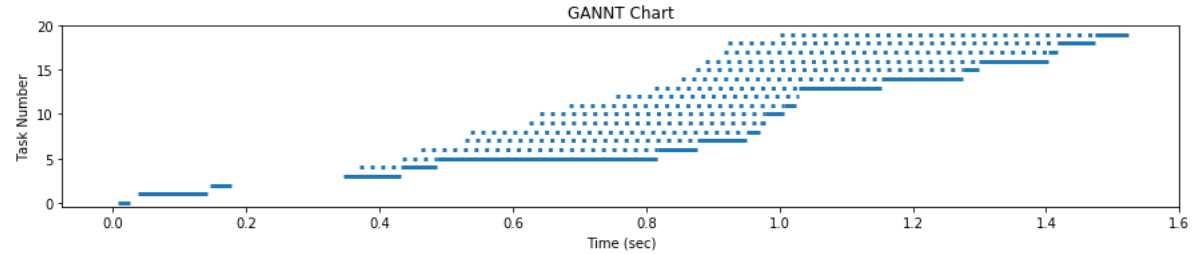


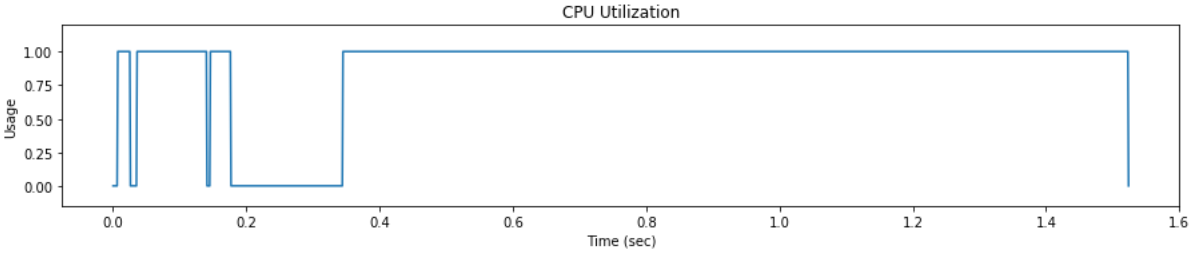

```
Number of Tasks:         20
Total Time:              4.0516
Mean Interarrival Time: 0.1889
Mean Wait Time:          0.2706
Mean Service Time:       0.1437
CPU Utilization:         0.7095
Maximum queue length:    5.0
Mean queue length:       1.3379
Std of queue length:     1.3784
[20, 4.051574839686111, 0.18891589756765198, 0.2705731570616766, 0.1437
3026957974938, 0.7095032192019666, 5.0, 1.337857847976308, 1.3783755663
480615]
```

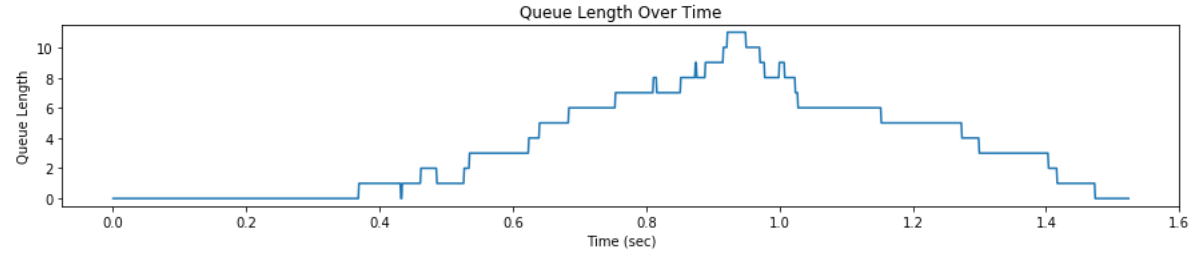
Part (C)

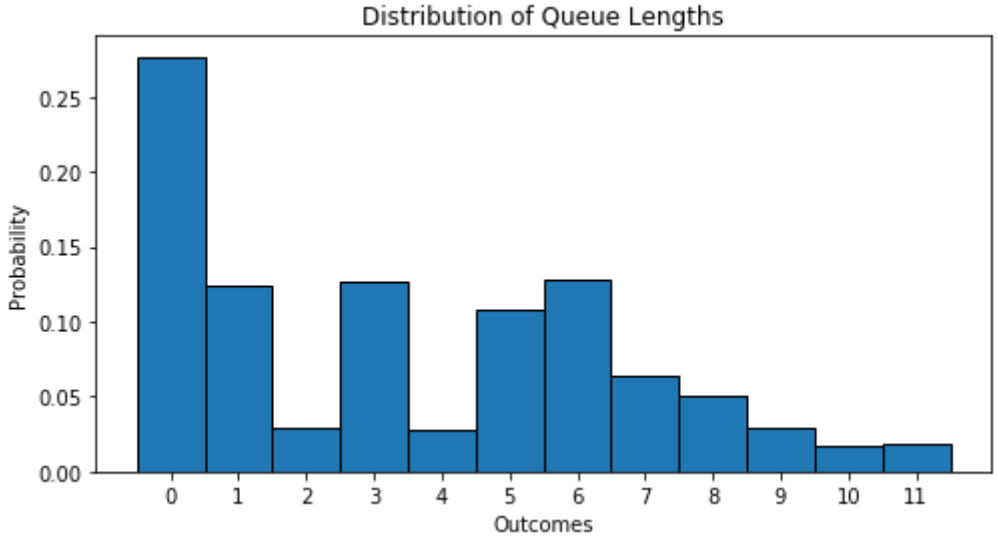
Now run the simulation with `lam` = 20.

In [10]:
```
lam = 20
stats = analyzeResults(run(getTaskList(num_tasks,lam,beta),"FIFO",False
))
```










```
Number of Tasks:         20
Total Time:              1.5243
Mean Interarrival Time: 0.05
Mean Wait Time:          0.2645
Mean Service Time:       0.0666
CPU Utilization:         0.8736
Maximum queue length:    11.0
Mean queue length:       3.4741
Std of queue length:     3.1273
[20, 1.524325761756567, 0.05000064683840767, 0.26450770562718035, 0.066
58063491167225, 0.8735748825099906, 11.0, 3.474098360655738, 3.12727052
0495144]
```

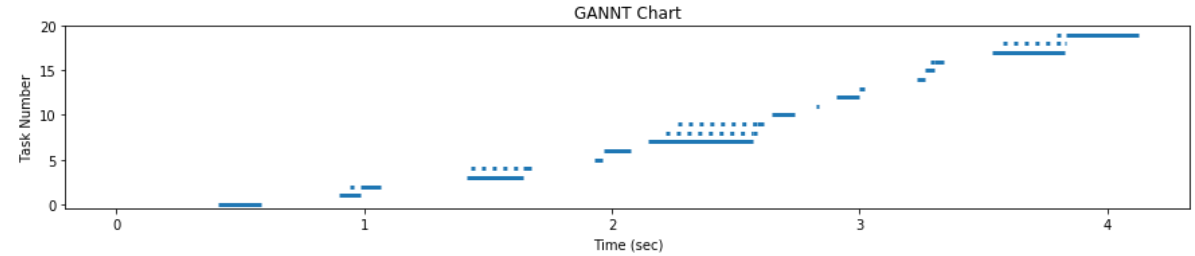
Part (D)

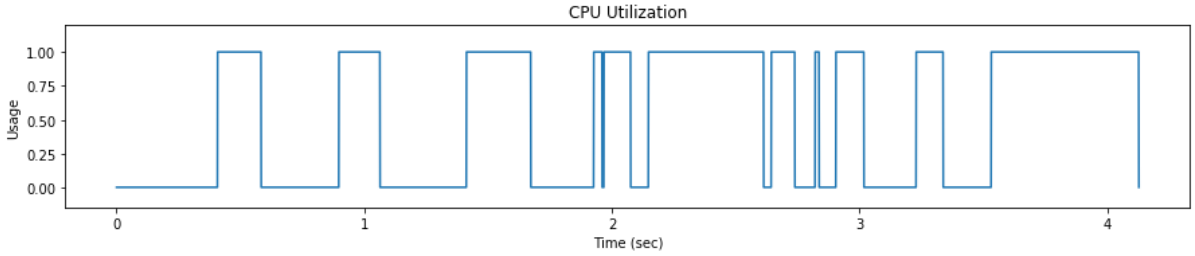
Now I would like you to also generate the same simulations, but using SJF (no need to give them all, just run them), and to run these various times to get a feel for how they behave, and then answer the following in a sentence or two:

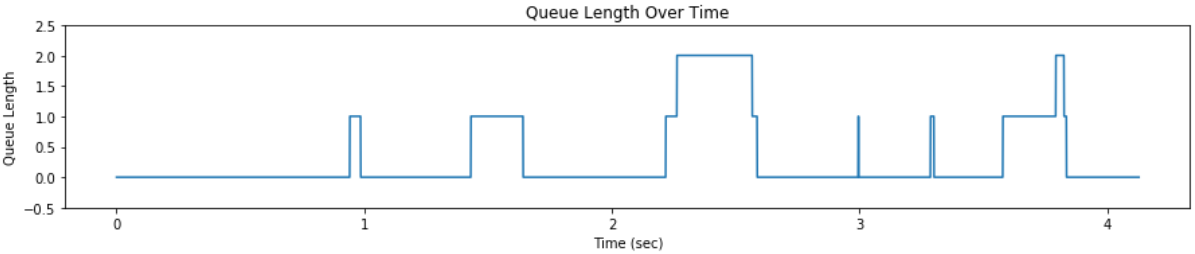
When the lam = 10, the arrival rate and departure rate are evenly matching, and we might expect that the system could deal with all the requests. But is this case? Look at your results and answer this question: **"When the rates are evenly matched, do the results look more like the overloaded case or the underloaded case? Does using FIFO vs SJF change this?"**

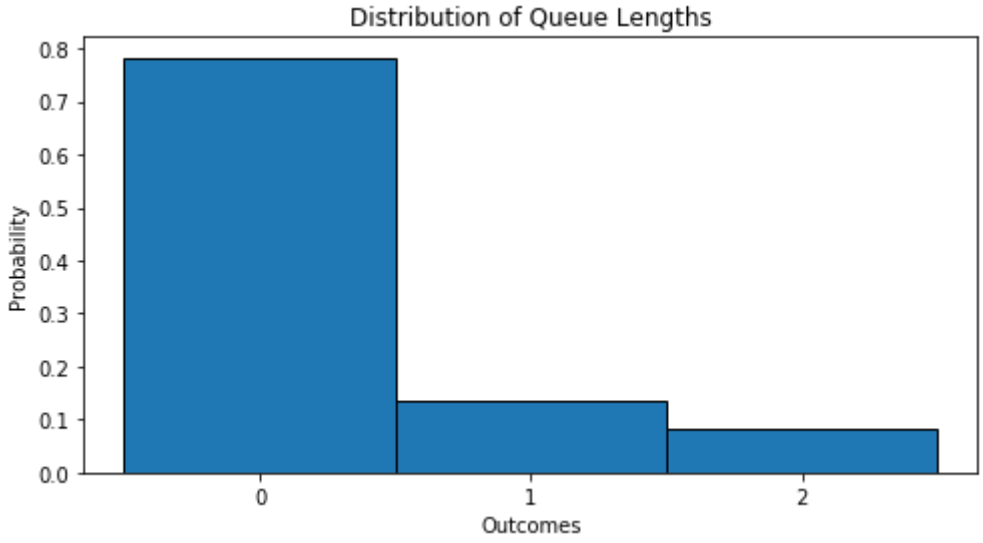
(Hint: don't worry about being too precise, I just want you to think about the issue a bit after observing the simulations.)

In [17]:
```
stats = analyzeResults(run(getTaskList(num_tasks,5,beta),"SJF",False))
stats = analyzeResults(run(getTaskList(num_tasks,10,beta),"SJF",False))
stats = analyzeResults(run(getTaskList(num_tasks,20,beta),"SJF",False))
```

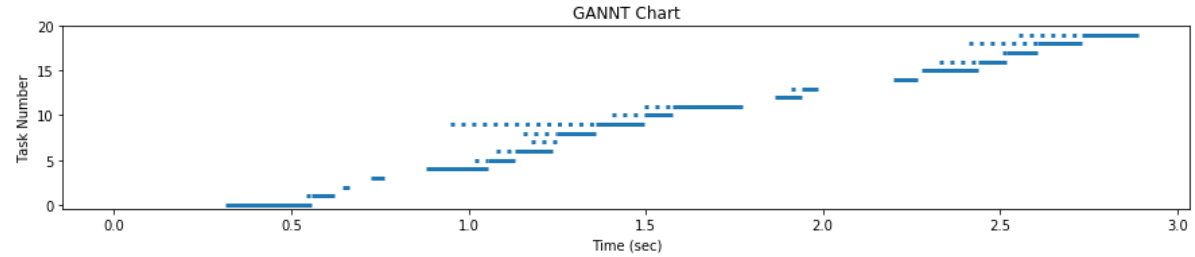


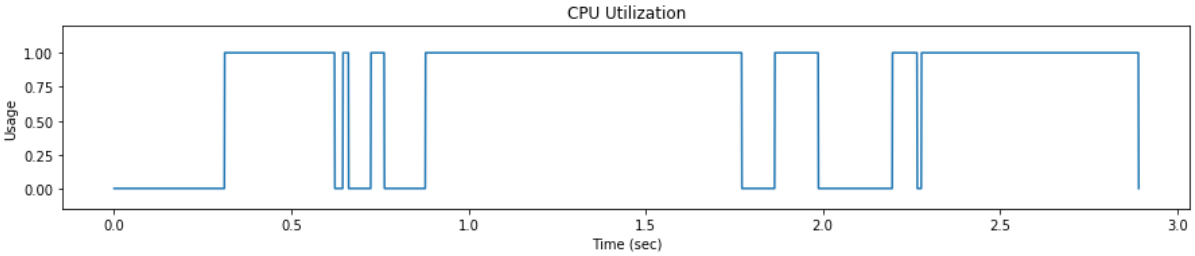


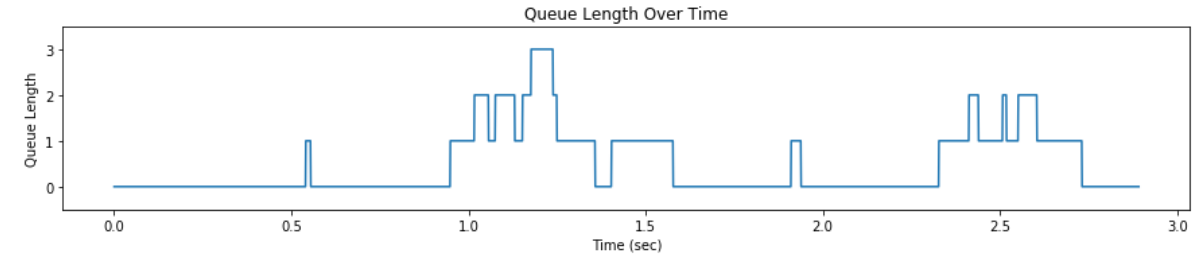


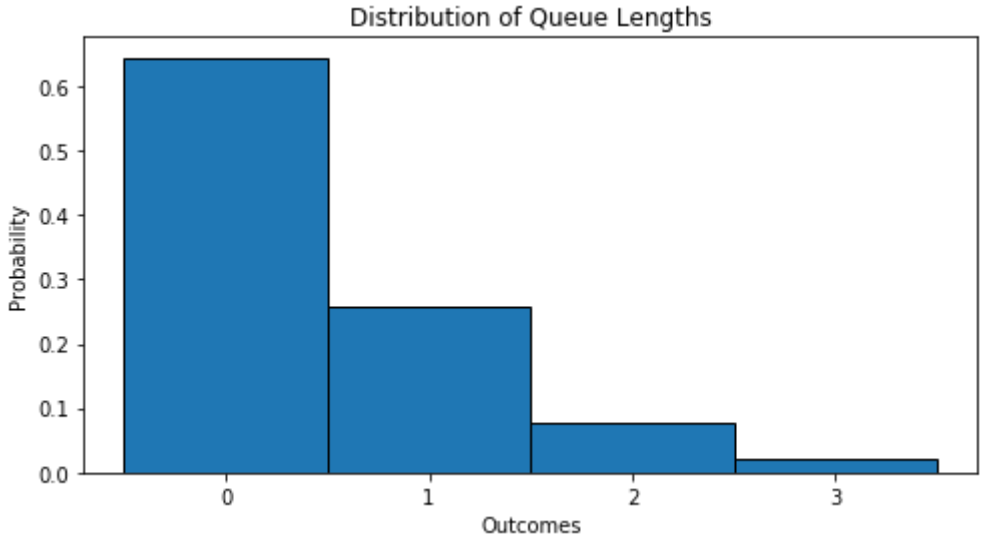



```
Number of Tasks:         20
Total Time:              4.1287
Mean Interarrival Time:  0.1897
Mean Wait Time:          0.0616
Mean Service Time:       0.1066
CPU Utilization:         0.5164
Maximum queue length:    2.0
Mean queue length:       0.2989
Std of queue length:     0.6106
```

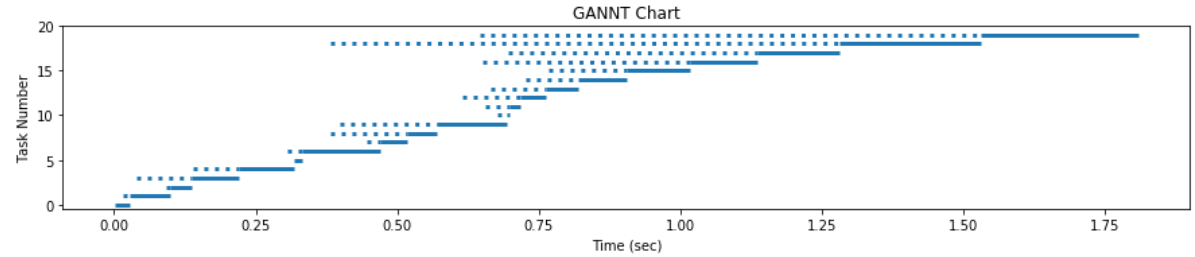


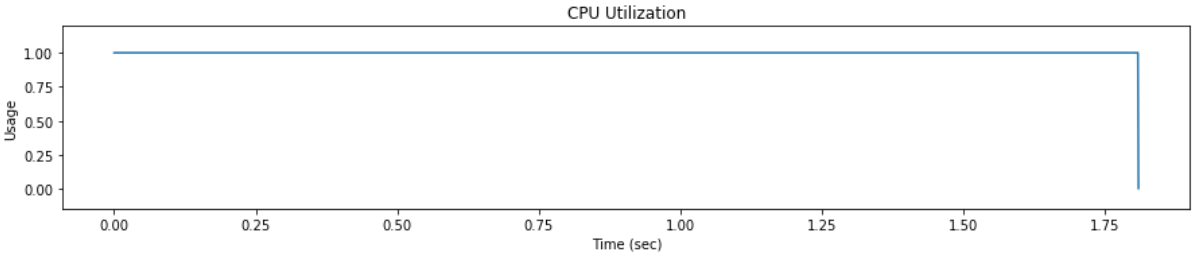


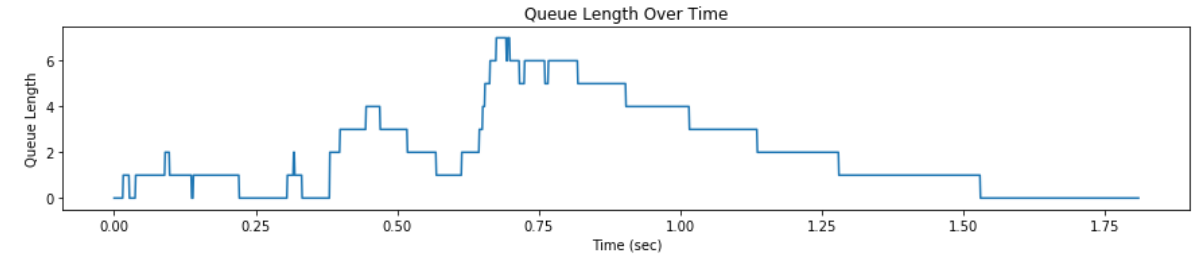


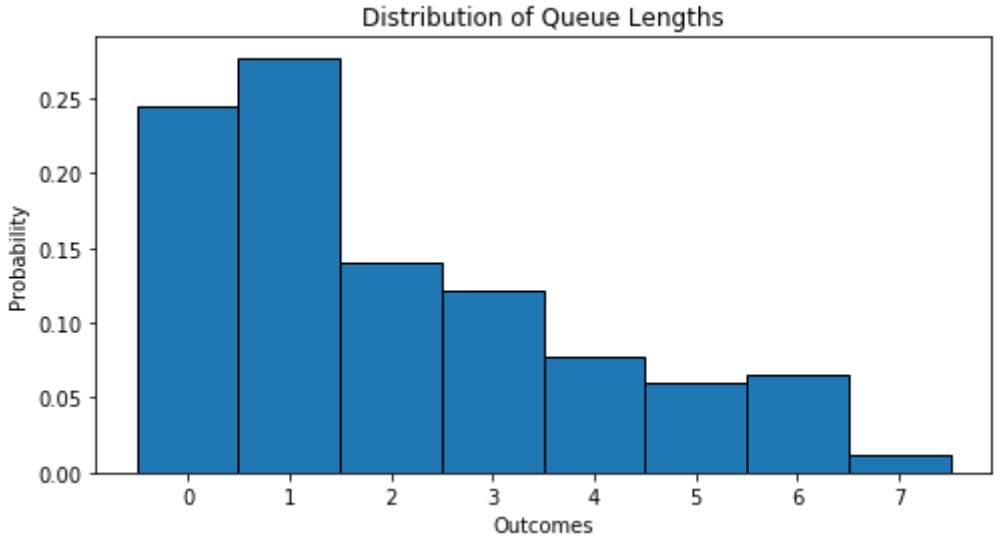



```
Number of Tasks:         20
Total Time:              2.8925
Mean Interarrival Time: 0.1276
Mean Wait Time:          0.0684
Mean Service Time:       0.103
CPU Utilization:         0.7124
Maximum queue length:    3.0
Mean queue length:       0.4749
Std of queue length:     0.7285
```










```
Number of Tasks:         20
Total Time:              1.8098
Mean Interarrival Time:  0.0323
Mean Wait Time:          0.1816
Mean Service Time:       0.0905
CPU Utilization:         0.9997
Maximum queue length:    7.0
Mean queue length:       2.0116
Std of queue length:     1.8922
```

Answer:

When the rates are evenly matched, the system is technically not yet in "overload" but still the tasks are interfering with each other, and the behavior we see in overload is starting to happen (long delays in the queue, chaotic behavior in queue, close to 100% CPU utilization after the system gets "saturated" with tasks.

I would say that for FIFO and lam = 10 it looks more like the overload case, but it is not completely certain.

Shortest job first seems to help this! It is hard to be sure, but to me, after running each one a bunch of times, it seems like in SJF the lam = 10 case is closer to the underloaded example.

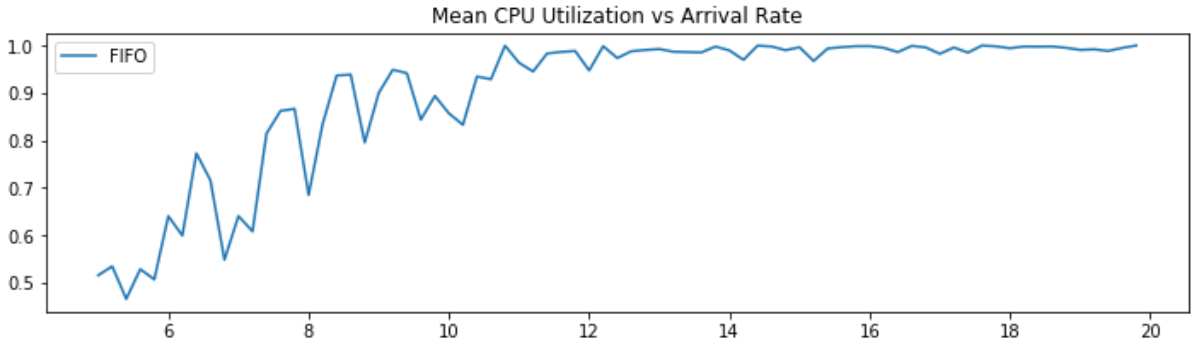
Problem Three

Now we are going to run multiple trials of our simulation with different values of lam, and graph and analyze the results. In this problem, we will see the effect of changing `lam` by graphing three different result values against `lam`.

Part (A)

Complete the following code template to graph the CPU Utilization against lam and observe that this increases to the overload point and then remains close to 1.0.

In [11]:
```
""" You may find the following useful:
numTasks = results[0]
totalTime = results[1]
meanInterarrivalTime = results[2]
meanWaitTime = results[3]
meanServiceTime = results[4]
cpuUtilization = results[5]
maxQueueLength = results[6]
meanQueueLength = results[7]
stdQueueLength = results[8]
"""

# Plot the statistic at results[numStat] against the sequence of lam val
ues in lams
# schedule is "FIFO" or "SJF" and titl is for the plot

def plotStat(numStat,numTasks,lams,beta,schedule,titl):

    # for each lam in lams, run the simulation and collect the statistic
results[numStat] in a list,
    # then plot these against lams to see the effect of arrival rate on
 this statistic

    meanList = []

    for lam in lams:
        #print(lam)
        taskList = getTaskList(numTasks,lam,beta)
        finished = run(taskList,schedule,False)
        results = analyzeResults(finished,False,False)
        meanList.append(results[numStat])


    plt.figure(figsize=(12, 3))
    plt.title(titl)
    plt.plot(lams,meanList,label=schedule)
    plt.legend()
    plt.show()

numTasks = 200
lams = list(np.arange(5,20,0.2))   # the different values of lam to use
 in this problem
beta = 0.1                         # mean service time of arriving tasks
-- do not change this parameter


plotStat(5,numTasks,lams,beta,schedule="FIFO",titl="Mean CPU Utilization
vs Arrival Rate")
```



Part (B)

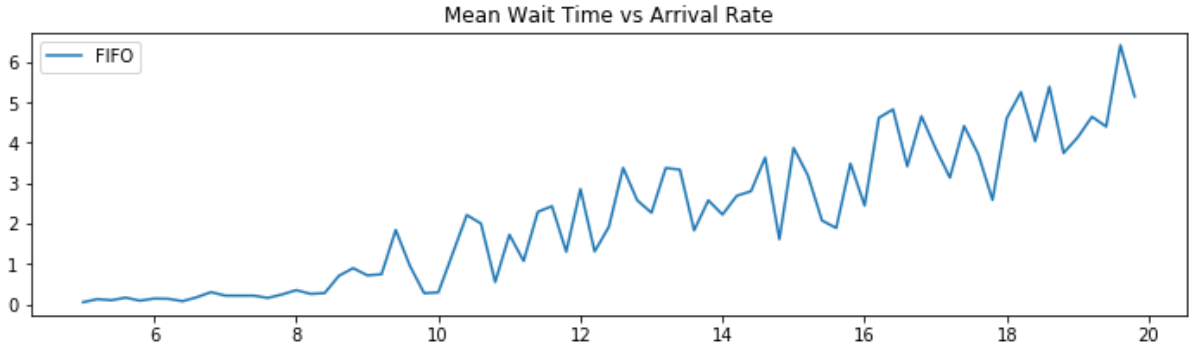
Now plot these parameters vs the arrival rate using FIFO:

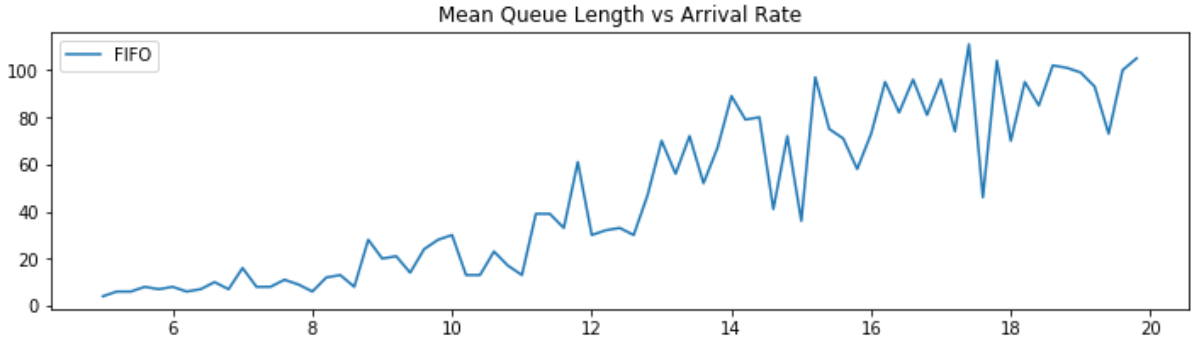
- Mean Wait Time
- Mean Queue Length
- Std of Queue Length

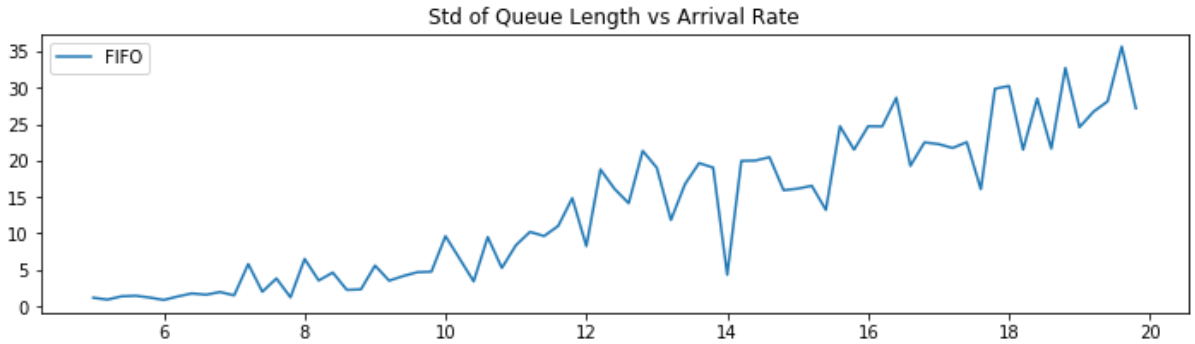
and answer the following question: "How do these behave as they exceed the overload point? Do changes happen at the overload point or before?"

I am just looking for general observations, nothing too deep, just a couple of sentences of what you observe.

In [12]:
```
plotStat(3,numTasks,lams,beta,schedule="FIFO",titl="Mean Wait Time vs Ar
rival Rate")
plotStat(6,numTasks,lams,beta,schedule="FIFO",titl="Mean Queue Length vs
Arrival Rate")
plotStat(8,numTasks,lams,beta,schedule="FIFO",titl="Std of Queue Length
 vs Arrival Rate")
```







Answer to (B):

Yes, the behavior we associated with overload--both increase in magnitude AND in the standard deviation--seems to start well before the overload point (lam = 10).

Problem Four

Now the question that might occur to you is: "What would happen if we use SJF in the previous problem"?

Compared with FIFO, SJF scheduling provides a shorter mean wait time (meaning, as a whole, jobs spend less time in the system) but may suffer from starvation (long jobs may wait a very long time).

We will only look at

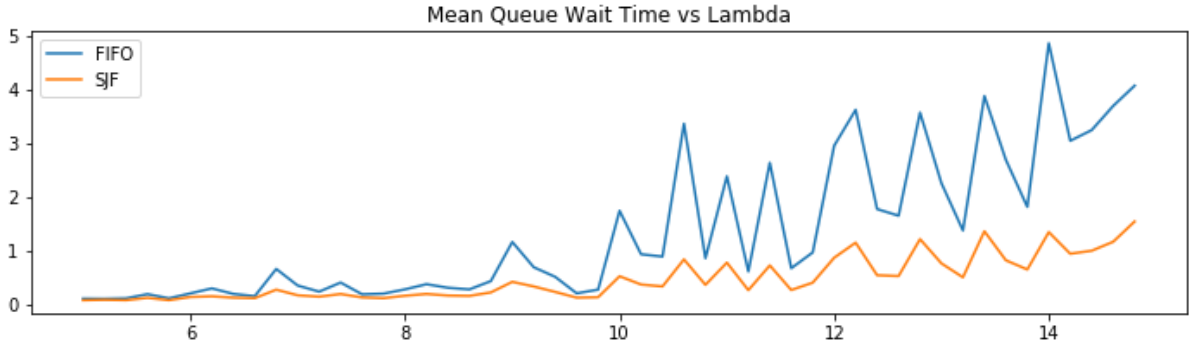
- CPU Utilization
- Mean Queue Length
- Std Queue Length

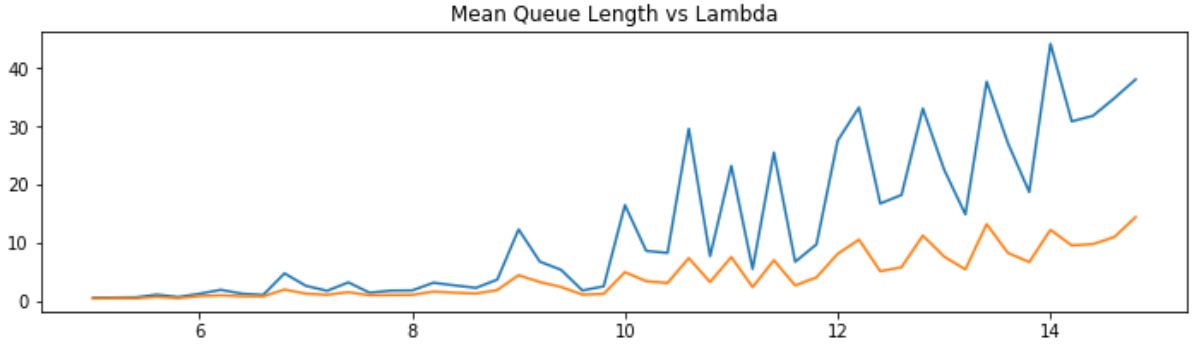
Part (A)

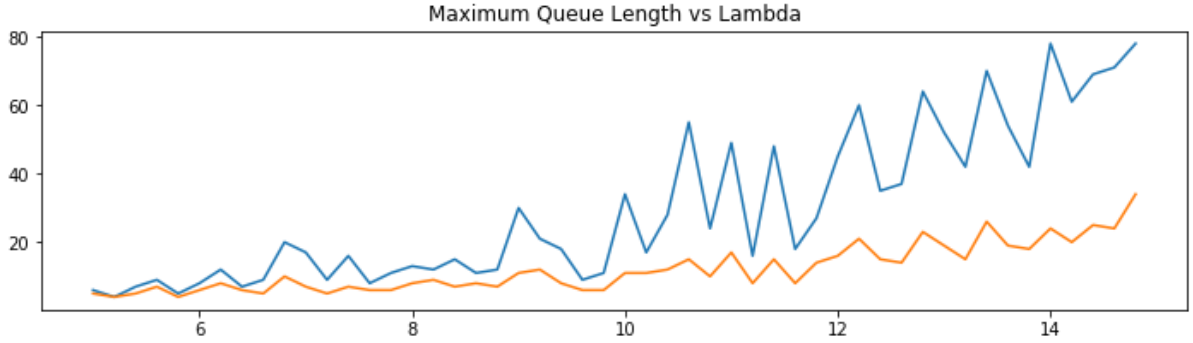
Repeat the experiment of the previous problem, but using FIFO and SJF and only for the three statistics just given. I would like you to plot FIFO and SJF on the same graph, in order to compare.

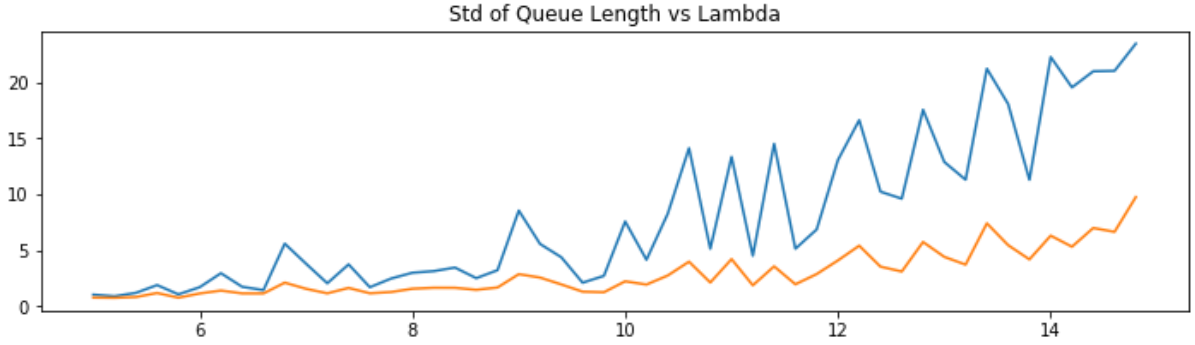
In [13]:
```
def plotStatistics2(numTasks,lams,beta):

    waitList = []
    cpuList = []
    maxList = []
    meanList = []
    stdList = []
    waitList2 = []
    cpuList2 = []
    maxList2 = []
    meanList2 = []
    stdList2 = []

    for lam in lams:
        #print(lam)
        task_list = getTaskList(numTasks,lam,beta)
        task_list2 = list(task_list)

        finished = run(task_list,"FIFO",False)
        results = analyzeResults(finished,False,False)

        finished2 = run(task_list2,"SJF",False)
        results2 = analyzeResults(finished2,False,False)

        waitList.append(results[3])
        cpuList.append(results[5])
        maxList.append(results[6])
        meanList.append(results[7])
        stdList.append(results[8])

        waitList2.append(results2[3])
        cpuList2.append(results2[5])
        maxList2.append(results2[6])
        meanList2.append(results2[7])
        stdList2.append(results2[8])

    plt.figure(figsize=(12, 3))
    plt.title("Mean Queue Wait Time vs Lambda")
    plt.plot(lams,waitList,label="FIFO")
    plt.plot(lams,waitList2,label="SJF")
    plt.legend()
    plt.show()

    plt.figure(figsize=(12, 3))
    plt.title("Mean Queue Length vs Lambda")
    plt.plot(lams,meanList)
    plt.plot(lams,meanList2)
    plt.show()

    plt.figure(figsize=(12, 3))
    plt.title("Maximum Queue Length vs Lambda")
    plt.plot(lams,maxList)
    plt.plot(lams,maxList2)
    plt.show()

    plt.figure(figsize=(12, 3))
```

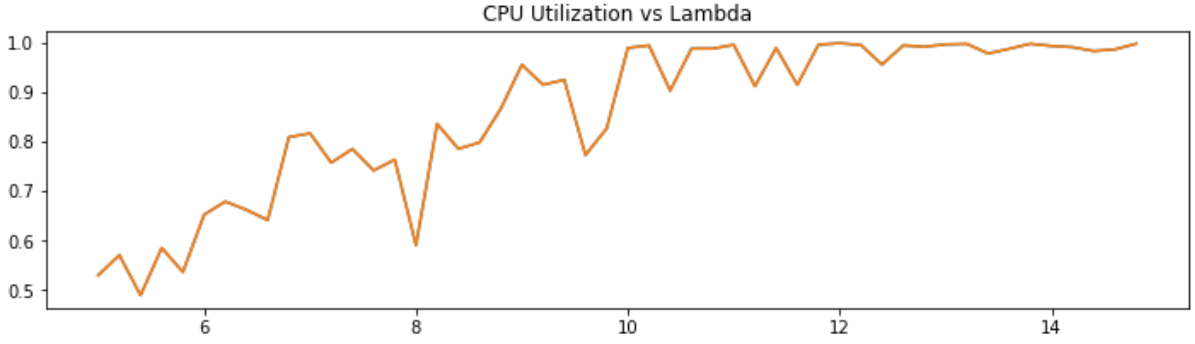
```
    plt.title("Std of Queue Length vs Lambda")
    plt.plot(lams,stdList)
    plt.plot(lams,stdList2)
    plt.show()

    plt.figure(figsize=(12, 3))
    plt.title("CPU Utilization vs Lambda")
    plt.plot(lams,cpuList)
    plt.plot(lams,cpuList2)
    plt.show()


numTasks = 200
lams = list(np.arange(5,15,0.2))   # the different values of lam to use
 in this problem
beta = 0.1                         # mean service time of arriving tasks
-- do not change this parameter

plotStatistics2(numTasks,lams,beta)
```


Mean Queue Wait Time vs Lambda
FIFO
SJF
0
1
2
3
4
5
6
8
10
12
14


Mean Queue Length vs Lambda
0
10
20
30
40
6
8
10
12
14


Maximum Queue Length vs Lambda
20
40
60
80
6
8
10
12
14


Std of Queue Length vs Lambda
0
5
10
15
20
6
8
10
12
14


CPU Utilization vs Lambda
0.5
0.6
0.7
0.8
0.9
1.0
6
8
10
12
14

Part (B)

Answer the following question: **"Describe what you see and what effect the SJF queue discipline has on these statistics. Do they both behave the same with regard to overload?"**

Hint: The CPU Utilization graph will look a little odd, and you may think you have a bug! Try graphing each curve separately and then together and you will understand what is going on.

Answer

Now it is clear that the SJF does a better job at managing this queue with respect to these measures. I designed my algorithm so that I am collecting FIFO and SJF data from the same tasklist for every lam, so you can see in each case how the curve is lower at pretty much every point, mitigating the effects of overload.

In particular, it really seems to mitigate the problem right at the overload point (lam = 10); it seems to delay the chaotic behavior until past the overload point.

For the CPU Utilization, the only time that FIFO vs SJF makes a different is when the queue is non-empty; both will empty out the queue and run all the tasks in the queue, they just do them in another order.

Problem Five

In this problem we are going to investigate **Little's Law**, which is described concisely in Wikipedia as follows:

> In queueing theory, a discipline within the mathematical theory of probability, Little's law is a theorem by John Little which states that the long-term average number L of customers in a stationary [i.e., not overloaded] system is equal to the long-term average effective arrival rate λ multiplied by the average time W that a customer spends in the system. Expressed algebraically the law is

$$L = \lambda \cdot W$$

Translating this into our parameters, and noting that the CPU utilization is the mean number of tasks in the CPU, this is equivalent to the following:

```
    (mean queue length + CPU utilization) = (Mean queue wait time + mean ser
vice time) / mean interarrival time
```

The importance of Little's Law is that it holds under a wide variety of different distributions, queueing disciplines, number of queues and servers, etc.

We are going to do the same experiment, but only plot the ratio of the left side of the equation with the right side, i.e.,

```
    littles_ratio = (mean queue length + CPU utilization) / ( (Mean queue wa
it time + mean service time) / mean interarrival time )
```

In [14]:

```
def plotStatistics3(numTasks,lams,beta):

    ratioList = []
    ratioList2 = []


    for lam in lams:
        #print(lam)
        task_list = getTaskList(numTasks,lam,beta)
        task_list2 = list(task_list)

        finished = run(task_list,"FIFO",False)
        results = analyzeResults(finished,False,False)

        finished2 = run(task_list2,"SJF",False)
        results2 = analyzeResults(finished2,False,False)

        littles_ratio = (results[7] + results[5]) / ( (results[3] + resu
lts[4]) / results[2] )
        littles_ratio2 = (results2[7] + results2[5]) / ( (results2[3] +
results2[4]) / results2[2] )

        ratioList.append(littles_ratio)
        ratioList2.append(littles_ratio2)

    plt.figure(figsize=(12, 3))
    plt.title("Little's Ratio for FIFO and SJF")
    plt.plot(lams,ratioList)
    plt.plot(lams,ratioList2)
    plt.show()



numTasks = 500
lams = list(np.arange(5,15,0.1))   # the different values of lam to use
 in this problem
beta = 0.1                         # mean service time of arriving tasks
-- do not change this parameter

plotStatistics3(numTasks,lams,beta)
```