University of Waterloo

ECE-358 COMPUTER NETWORKS

Course Fall 2023

LAB 1

Group 151

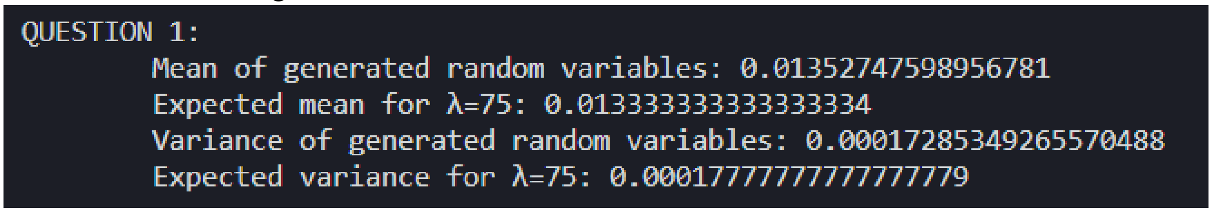
József IVÁN GAFO (WatId: 21111635, jivangaf)

Sonia NAVAS RUTETE (WatId: 21111397, srutete)

TABLE OF CONTENTS

QUESTION 1

For calculating the mean and variance of 1000 random exponential variables, we have generated them by using the formula: x=-(1/lambda)Ln(1-U) where U is a random number (0,1), in a 1000 times loop. After that we do the mean and variance using numpy python library (makes it easier). According to calculating the expected ones we did it manually: mean = 1/lambda and variance = 1/ lambda^2
In the following example we can observe that the expected mean and variance are similar to the ones we have generated.



```
QUESTION 1:
        Mean of generated random variables: 0.01352747598956781
        Expected mean for λ=75: 0.013333333333333334
        Variance of generated random variables: 0.00017285349265570488
        Expected variance for λ=75: 0.00017777777777777779
```

QUESTION 2

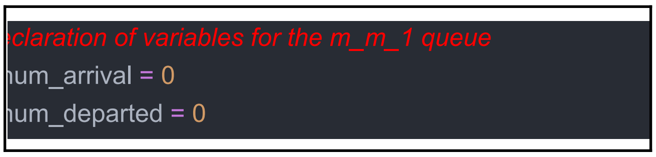
Explain the code:

The code for this m_m_1 queue can be found in the annex of this document.
The functions that we will use are self.m_m_1_queue(...) that is in charge of generating the queue and self.__generate_exp_distribution(...) in charge of generating a number from the exponential distribution of poisson (that was also used on the exercise1).

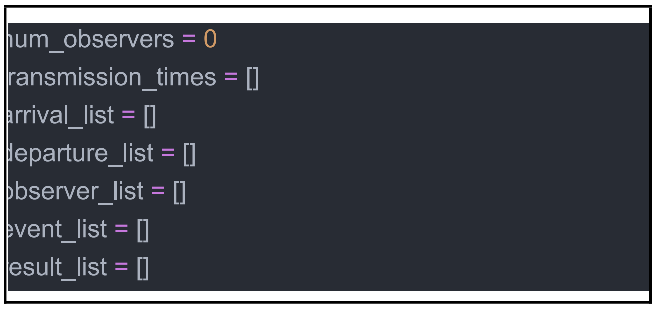
a) Inputs:

This function takes as input the avg_len that is the average length of the packets, then we have trans_rate that is the transmission rate that our queue has and lambda_par is the parameter lambda of the poisson distribution.

b) We define variables for the queue



```
claration of variables for the m_m_1 queue
um_arrival = 0
um_departed = 0
```

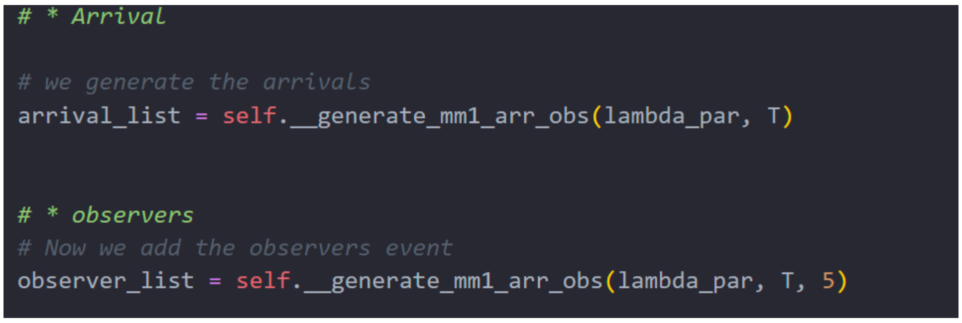


```
um_observers = 0
ransmission_times = []
rrival_list = []
eparture_list = []
bserver_list = []
vent_list = []
esult_list = []
```

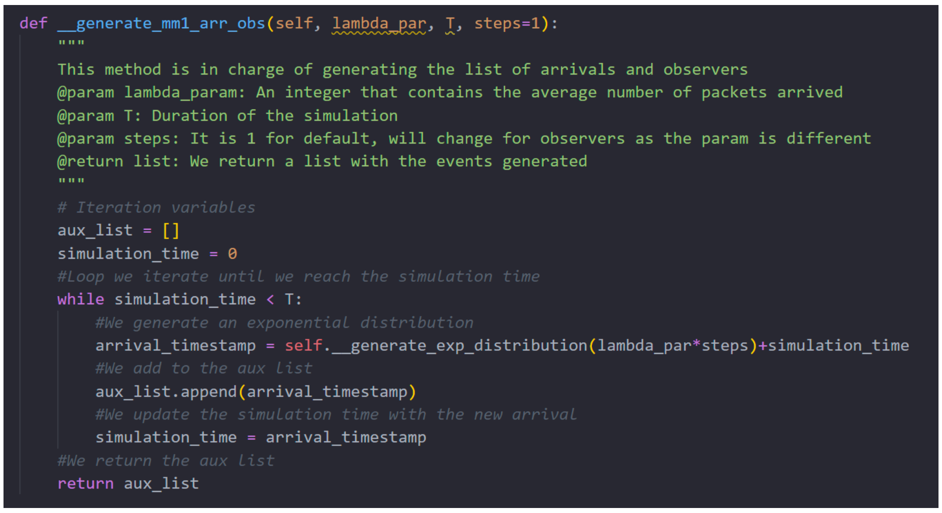
In the code we first define the variables for our queue

- We first have 3 counters that are num_arrival, num_departed and num_observers that are in charge of counting the events that are occurring when we start to analyze event per event.
- Then we have 3 lists that are arrival_list, departure_list and observer_list. In those 3 lists we will store all the events that are generated on the simulation.
- result_list is a list where we store all the events unsorted.
- event_list is a list where we will sort all the events.

c) We generate the arrivals and observations times



```
# * Arrival

# we generate the arrivals
arrival_list = self.__generate_mm1_arr_obs(lambda_par, T)


# * observers
# Now we add the observers event
observer_list = self.__generate_mm1_arr_obs(lambda_par, T, 5)
```

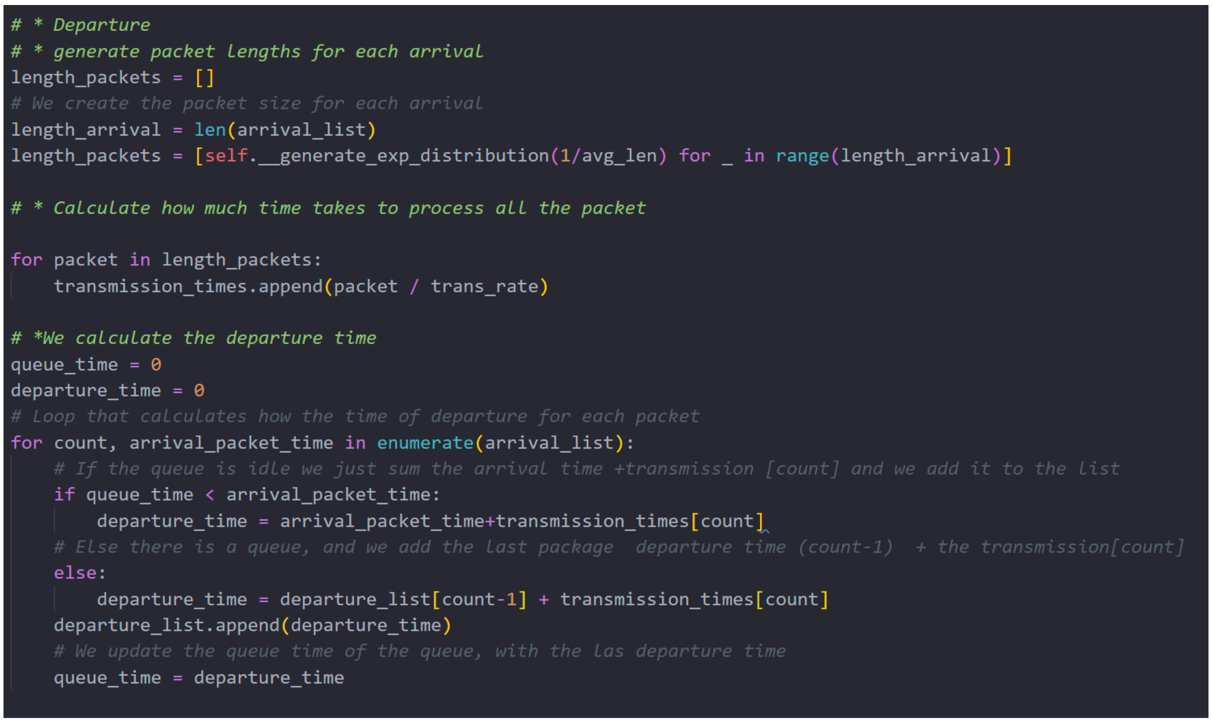


```
def __generate_mm1_arr_obs(self, lambda_par, T, steps=1):
    """
    This method is in charge of generating the list of arrivals and observers
    @param lambda_param: An integer that contains the average number of packets arrived
    @param T: Duration of the simulation
    @param steps: It is 1 for default, will change for observers as the param is different
    @return list: We return a list with the events generated
    """
    # Iteration variables
    aux_list = []
    simulation_time = 0
    #Loop we iterate until we reach the simulation time
    while simulation_time < T:
        #We generate an exponential distribution
        arrival_timestamp = self.__generate_exp_distribution(lambda_par*steps)+simulation_time
        #We add to the aux list
        aux_list.append(arrival_timestamp)
        #We update the simulation time with the new arrival
        simulation_time = arrival_timestamp
    #We return the aux list
    return aux_list
```

For generating the observations and arrivals lists we will use the function ***generate_mm1_arr_obs(...)*** , that is in charge of generating a list of random exponential distributions for a given ***simulation_time ,T and lambda***. We first create the iteration variables for the while, then we enter the while loop that will iterate until we reach the max time of the simulation. Then, we generate an exponential distribution (the function only returns the formula explained on exercise 1) plus the previous simulation time and we update on

arrival_timestamp that we will append it on ***aux_list*** and update the ***simulation_time***. And when the loop finishes we return the aux_list that we will later assign to ***arrival_list*** and ***observer_list.*** We also know that those list are ordered because we have added them in order of simulation time.

d) We generate the departures



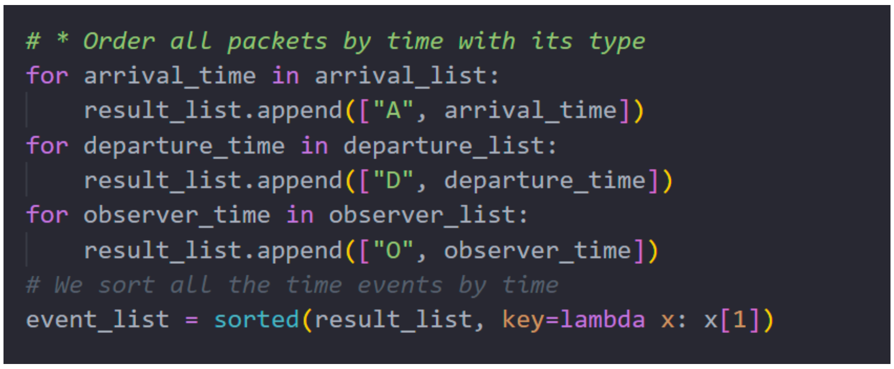
```python
# * Departure
# * generate packet lengths for each arrival
length_packets = []
# We create the packet size for each arrival
length_arrival = len(arrival_list)
length_packets = [self.__generate_exp_distribution(1/avg_len) for _ in range(length_arrival)]

# * Calculate how much time takes to process all the packet

for packet in length_packets:
    transmission_times.append(packet / trans_rate)

# *We calculate the departure time
queue_time = 0
departure_time = 0
# Loop that calculates how the time of departure for each packet
for count, arrival_packet_time in enumerate(arrival_list):
    # If the queue is idle we just sum the arrival time +transmission [count] and we add it to the list
    if queue_time < arrival_packet_time:
        departure_time = arrival_packet_time+transmission_times[count]
    # Else there is a queue, and we add the last package  departure time (count-1)  + the transmission[count]
    else:
        departure_time = departure_list[count-1] + transmission_times[count]
    departure_list.append(departure_time)
    # We update the queue time of the queue, with the las departure time
    queue_time = departure_time
```

Now we will calculate the departure time for each arrival. First we will generate exponential random variables as many arrivals that we have with **lambda=1/average length packet** and store it on ***length_packets***, then we calculate the transmission time (packet length / transmission rate) and we save it on ***transmission_times***. Where the pos x on transmission time list is the same as the arrivals.

After calculating all transmission times, we will calculate the departure time for each arrival packet. For this we will use 2 iteration variables that are ***queue_time*** that is in charge of having the information of the actual time of the simulation and ***departure_time*** that is in charge of holding the information of the previous departure.

We will use a for loop that will iterate on the arrival_list elements. Inside the loop we do a conditional where we check if the ***queue_time < arrival_packet_time*** if is true it means that the queue is idle and the ***departure_time*** is ***arrival_packet_time+transmission_times[count]*** else it means that the queue is not idle and the ***departure_time*** is simply the departure time of the ***previous packet + the transmission time***.

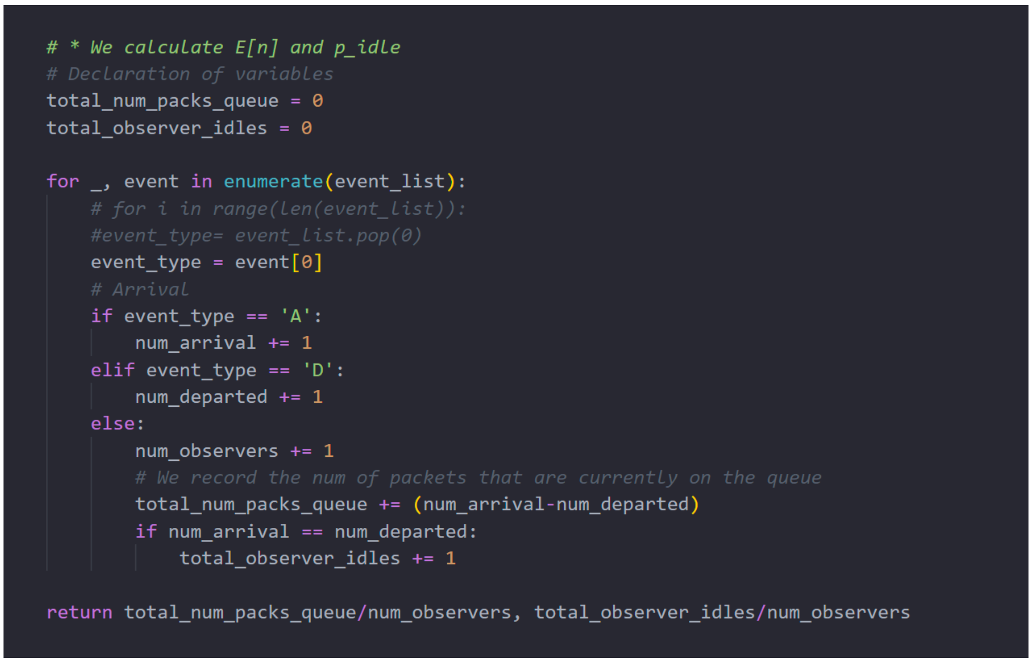
e) Order packets by type and time



```
# * Order all packets by time with its type
for arrival_time in arrival_list:
    result_list.append(["A", arrival_time])
for departure_time in departure_list:
    result_list.append(["D", departure_time])
for observer_time in observer_list:
    result_list.append(["O", observer_time])
# We sort all the time events by time
event_list = sorted(result_list, key=lambda x: x[1])
```

In this part we first append to the ***result_list*** the arrivals on the following structure [event_type,time], event type means the event type that can be “A” for arrival, “D” for departure and “O” for observer. Once we add all of them on the result list we sort them and save them on ***event_list*** that contains all the events in order.

f) Calculate E[n] and p_idle



```
# * We calculate E[n] and p_idle
# Declaration of variables
total_num_packs_queue = 0
total_observer_idles = 0

for _, event in enumerate(event_list):
    # for i in range(len(event_list)):
    #event_type= event_list.pop(0)
    event_type = event[0]
    # Arrival
    if event_type == 'A':
        num_arrival += 1
    elif event_type == 'D':
        num_departed += 1
    else:
        num_observers += 1
        # We record the num of packets that are currently on the queue
        total_num_packs_queue += (num_arrival-num_departed)
        if num_arrival == num_departed:
            total_observer_idles += 1

return total_num_packs_queue/num_observers, total_observer_idles/num_observers
```

This is the last part of the mm1 queue and is in charge of calculating the total average number of packets in the queue and the total average of packets on the queue.

First, we will define the iteration variables that are ***total_num_packs_queue*** in charge of having the total number of packets in the queue on an observer point and ***total_observer_idles*** in charge of having the total number of packets that are idle on an observer point.

Then we do a for loop that will iterate through the ***event_list***. If the ***event_type*** is an arrival (“A”) then we sum the counter of arrivals +1, if it is a departure then we sum the counter of departures +1.

If is an observer(“O”) we sum the observer count +1 and we sum to the ***total_num_packs_queue*** the ***number of arrivals - the number of departures*** and if the

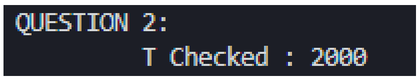
number of arrivals == the number of departures we sum 1 to the ***total_number_of_idles***.

When the for loop is finished we finished the simulation of the mm1 queue and we return on pso[0] the **average number of packets in the queue** (E[n]) by calculating ***total_num_packs_queue/ num_observers*** and on pos[o] we return **p_idle** that is ***calculated by total_observers_idle/num_observers***

Which simulation T do we take?

To know the simulation T we build a python function called check_T that is in charge of running the mm1 queue with different T such as: 2T, 3T… and prints you the best simulation time. It selects it in the following way: it makes the difference between the results for T and T+1 and checks if it is included in the range for 5% of difference. In the case it is not included we will continue until T is included, else we will try with the next one if it is included we choose the lowest difference.

We observed that it gives you different simulation times because it depends on the exponential distributions but the most simulation times that prints are between 2T and 3T. So for the following exercises we will use 2T because it makes our simulations much faster.

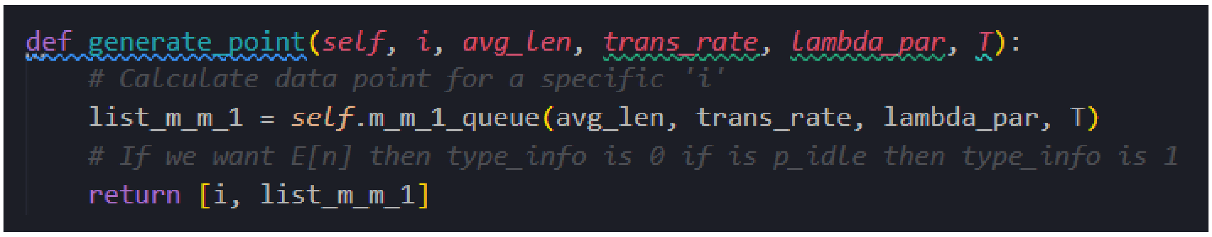


```
QUESTION 2:
        T Checked : 2000
```

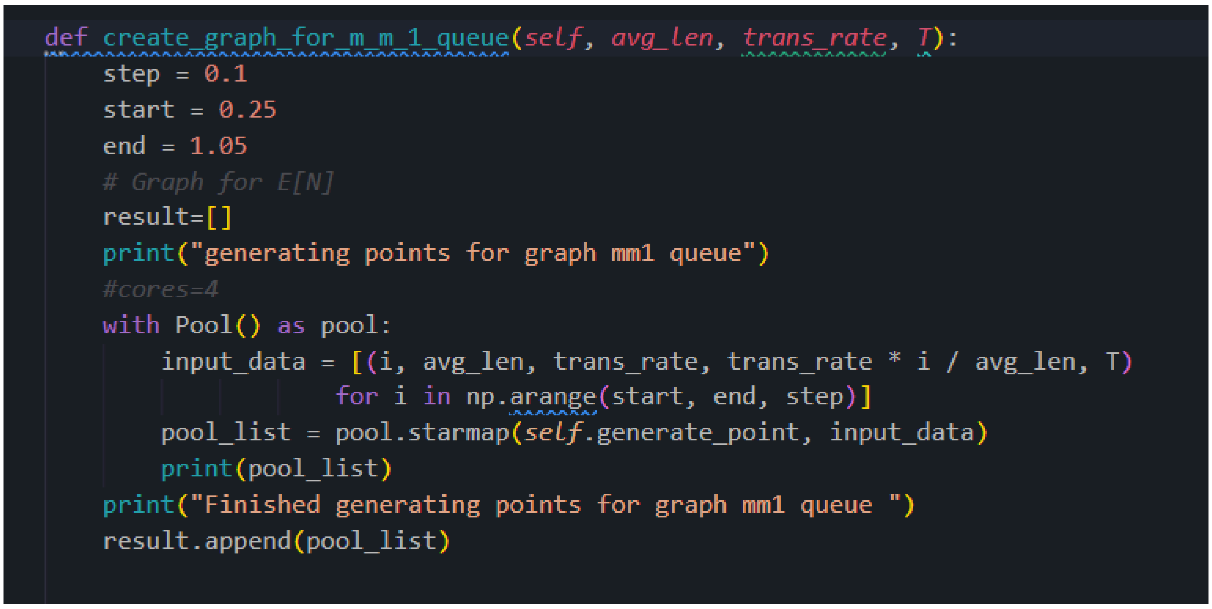
QUESTION 3

First of all we have taken the parameters L=2000 bits and C=10^6 bps. After doing the simulation, explained before we generate the points for p, which goes from 0.25 to 0.95 with 0.1 steps. p only changes the lambda parameter that we insert in the mm1 function: the new lambda parameter will be lambda = C * p / L. Our code is just a loop where we generate the points for each p value, adding the E[N] and Pidle to a list.

As we can see below, we got the function create graph that calls generate_points 8 times for getting the points. We use pool for code efficiency(to have a function per core in our computer, so it runs faster).



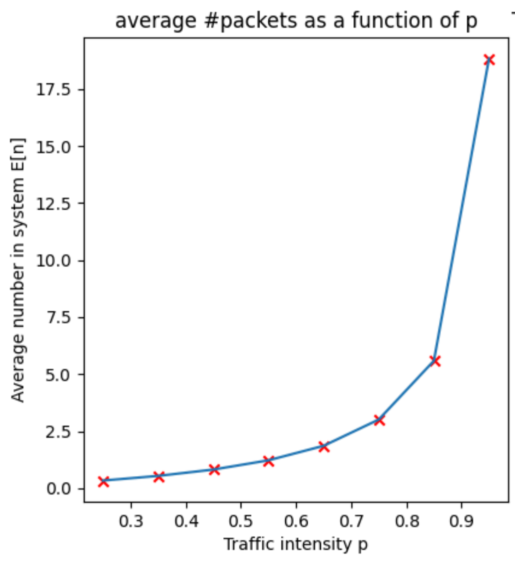
```python
def generate_point(self, i, avg_len, trans_rate, lambda_par, T):
    # Calculate data point for a specific 'i'
    list_m_m_1 = self.m_m_1_queue(avg_len, trans_rate, lambda_par, T)
    # If we want E[n] then type_info is 0 if is p_idle then type_info is 1
    return [i, list_m_m_1]
```

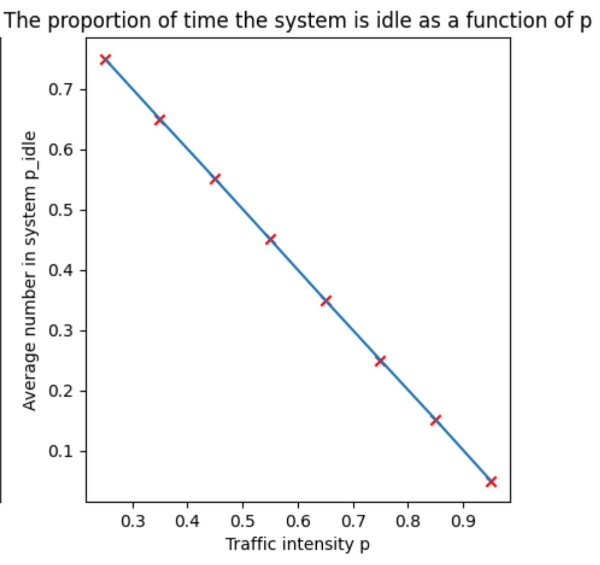


```
def create_graph_for_m_m_1_queue(self, avg_len, trans_rate, T):
    step = 0.1
    start = 0.25
    end = 1.05
    # Graph for E[N]
    result=[]
    print("generating points for graph mm1 queue")
    #cores=4
    with Pool() as pool:
        input_data = [(i, avg_len, trans_rate, trans_rate * i / avg_len, T)
                        for i in np.arange(start, end, step)]
        pool_list = pool.starmap(self.generate_point, input_data)
        print(pool_list)
    print("Finished generating points for graph mm1 queue ")
    result.append(pool_list)
```

If we run our code once, we get these values, we can appreciate that when p increases, E[N] does the same, not as Pidle. Which means that when we increase p the packets in the buffer increase, and the proportion of time the queue is idle will be lower.

p	0.25	0.35	0.45	0.55	0.65	0.75	0.85	0.95
E[N]	0.3334	0.5376	0.8150	1.2204	1.8556	3.0079	5.5834	18.8348
Pidle	0.7501	0.6500	0.5509	0.4505	0.3496	0.2503	0.1516	0.0505





QUESTION 4

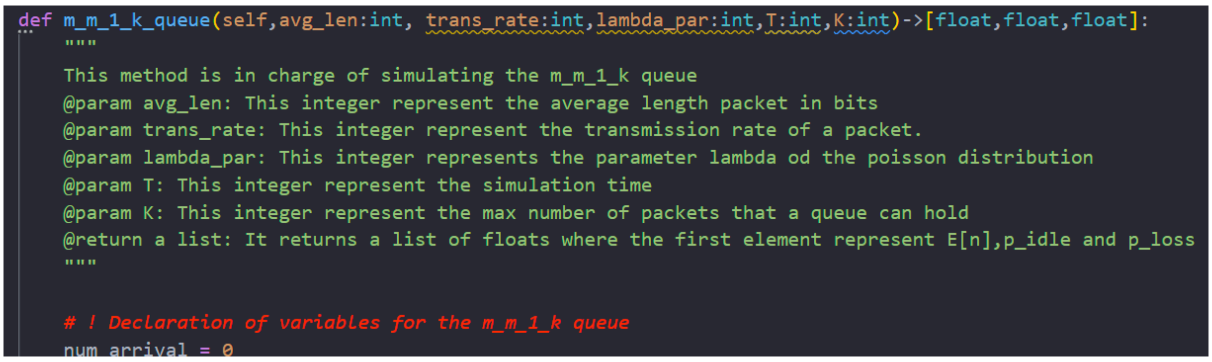
For p= 1.2 we can observe that the traffic intensity, E(N) is incredibly increased and the Pidle is considerably decreased. As we can appreciate in the image of the code and the compilation.

```
   E, pidle = a.m_m_1_queue(avg_packet_length, trans_rate, trans_rate * 1.2 / avg_packet_length, 2*T)
       print("\nFor p = 1.2, the value of E[N] = "+ str(E) + " and the value of pidle =" + str(pidle))

   EXECUTION:

   For p = 1.2, the value of E[N] = 101034.26367365489 and the value of pidle =7.163822629082921e-06
```

QUESTION 5

Explain the code:

a) Inputs:

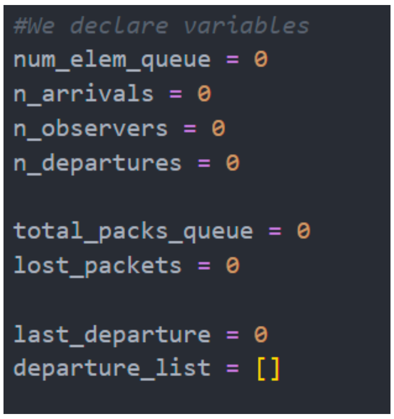


```
def m_m_1_k_queue(self,avg_len:int, trans_rate:int,lambda_par:int,T:int,K:int)->[float,float,float]:
    """
    This method is in charge of simulating the m_m_1_k queue
    @param avg_len: This integer represent the average length packet in bits
    @param trans_rate: This integer represent the transmission rate of a packet.
    @param lambda_par: This integer represents the parameter lambda od the poisson distribution
    @param T: This integer represent the simulation time
    @param K: This integer represent the max number of packets that a queue can hold
    @return a list: It returns a list of floats where the first element represent E[n],p_idle and p_loss
    """

    # ! Declaration of variables for the m_m_1_k queue
    num_arrival = 0
```

Here we define the main function to simulate the m m 1 k queue. This function to run needs 5 inputs.

- avg_len: it tells you the average length of a packet in bits
- trans_rate: it tells you the transmission rate of the simulation
- lambda_par: represents the parameter lambda of the exponential distribution
- T: It represents the simulation time of the queue
- K: Represents the max number of packets the queue can have on a given moment

b) We define the variables for the simulation:



```
#We declare variables
num_elem_queue = 0
n_arrivals = 0
n_observers = 0
n_departures = 0

total_packs_queue = 0
lost_packets = 0

last_departure = 0
departure_list = []
```

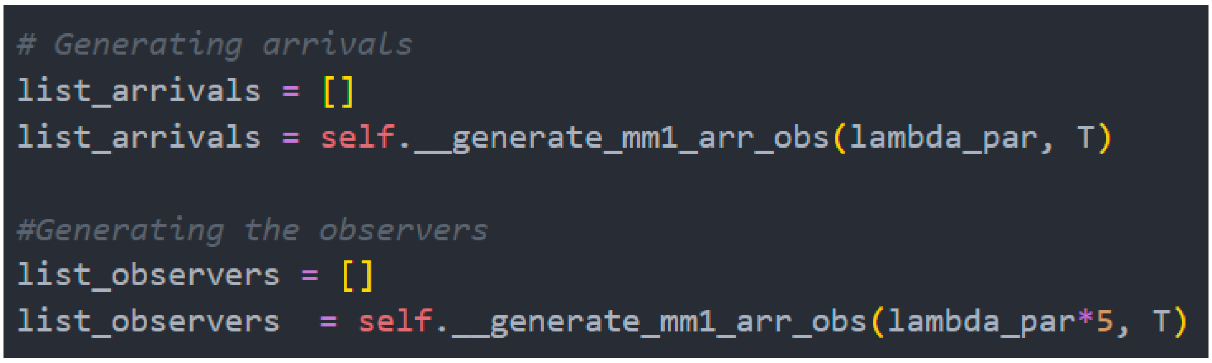
n_arrivals, n_observers and n_departures, lost_packets are counters that will be helpful for the observers on the last loop of our code.

num_elem_queue tells us the number of elements in the queue

total_packs_queue tells how many packets are in the queue in all the observers event

departure_list is the list where we will store the departures events on the last part of the code.

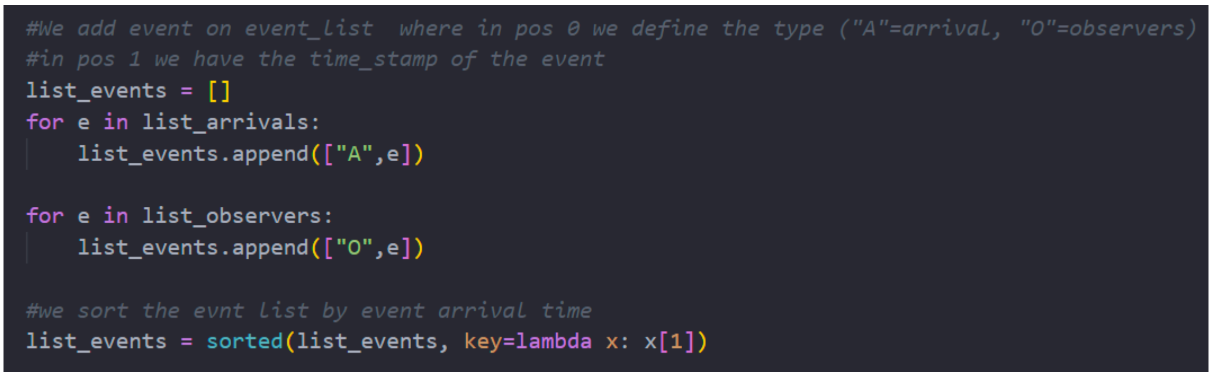
c) We generate the observers and arrivals



```
# Generating arrivals
list_arrivals = []
list_arrivals = self.__generate_mm1_arr_obs(lambda_par, T)

#Generating the observers
list_observers = []
list_observers  = self.__generate_mm1_arr_obs(lambda_par*5, T)
```

This part is the same as in question 2, where we generate the arrivals and observers

d) We order packets by type and sort them

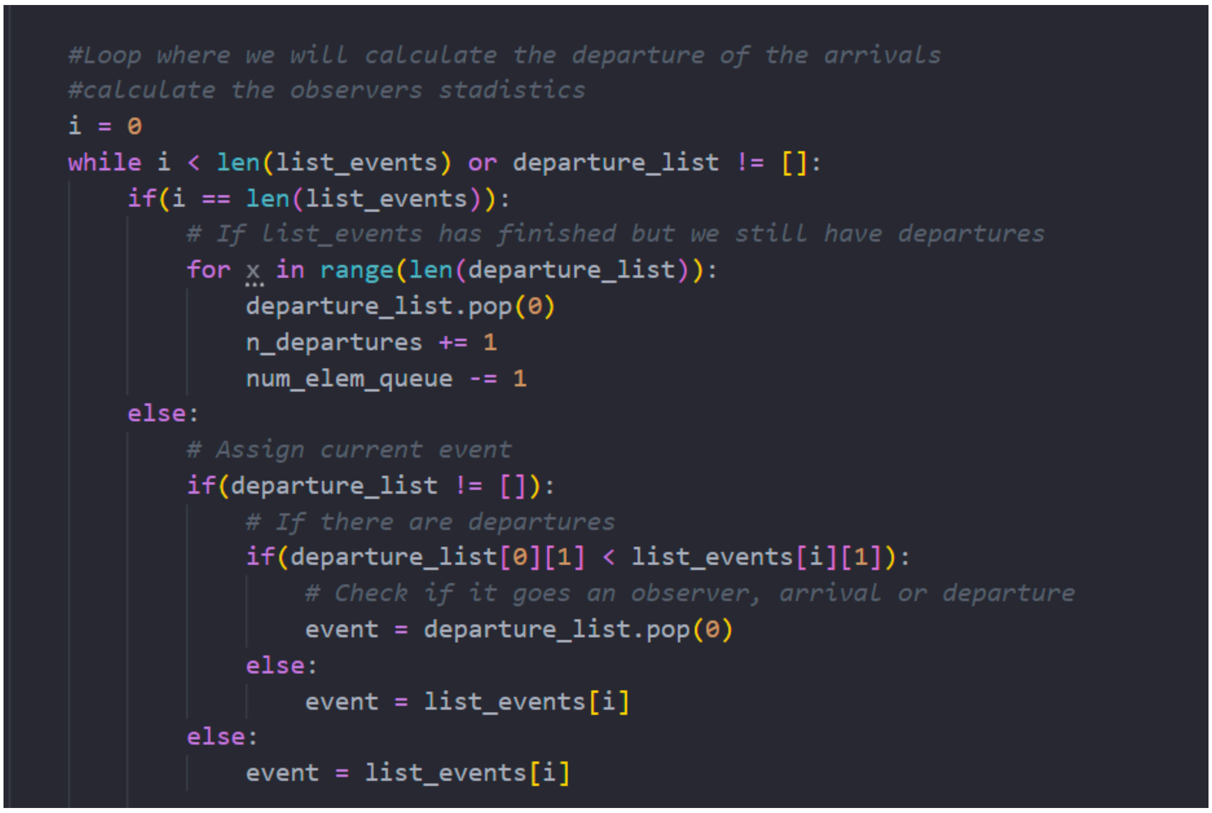


```
#We add event on event_list  where in pos 0 we define the type ("A"=arrival, "O"=observers)
#in pos 1 we have the time_stamp of the event
list_events = []
for e in list_arrivals:
    list_events.append(["A",e])

for e in list_observers:
    list_events.append(["O",e])

#we sort the evnt list by event arrival time
list_events = sorted(list_events, key=lambda x: x[1])
```

This part is the same as in question 2, but we only sort the arrivals and observers

e) We generate departures and calculate E[n],p_idle and p_loss

Now we enter in a loop where we will calculate all the events and adding departures events as the simulation is going on.

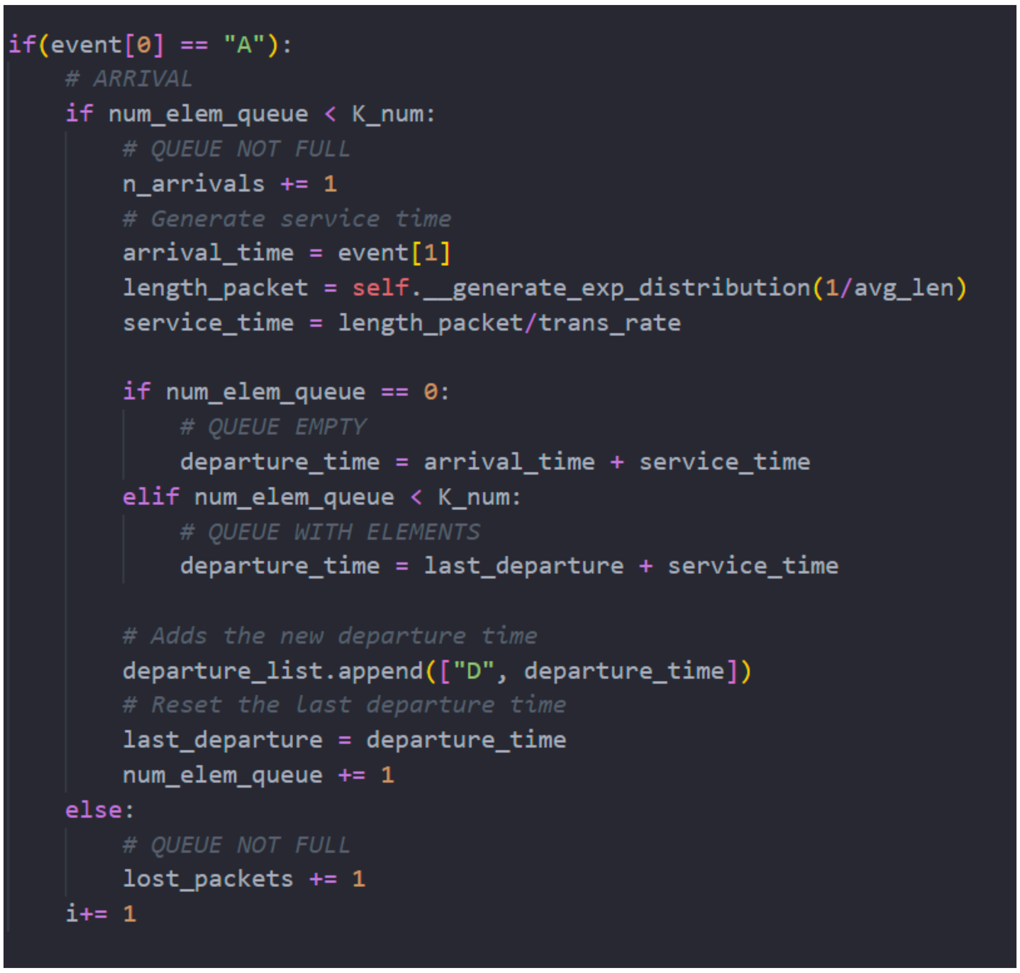


```python
#Loop where we will calculate the departure of the arrivals
#calculate the observers stadistics
i = 0
while i < len(list_events) or departure_list != []:
    if(i == len(list_events)):
        # If list_events has finished but we still have departures
        for x in range(len(departure_list)):
            departure_list.pop(0)
            n_departures += 1
            num_elem_queue -= 1
    else:
        # Assign current event
        if(departure_list != []):
            # If there are departures
            if(departure_list[0][1] < list_events[i][1]):
                # Check if it goes an observer, arrival or departure
                event = departure_list.pop(0)
            else:
                event = list_events[i]
        else:
            event = list_events[i]
```

We will iterate until i<len(list_events) meaning that we will iterate until the event_list is empty or the departure_list is not empty.

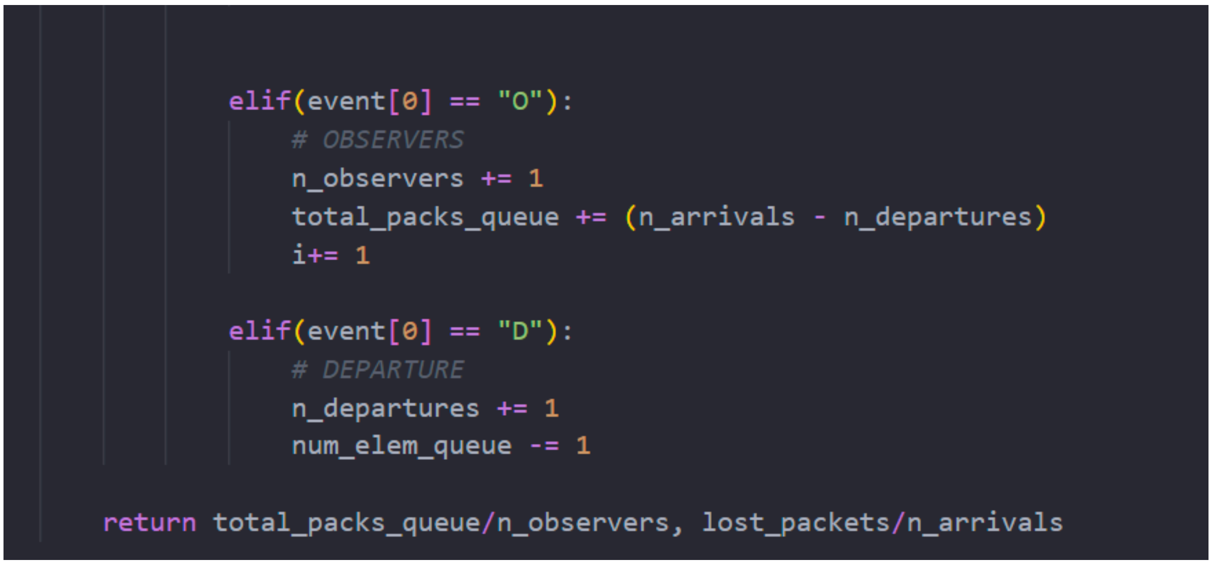
Then we do a conditional, and if the event_list is empty there are only departures events in the queue and we start to pop them and update counters.

Else the event_list is not empty, then we check if there is a departure that comes first than the actual event on the event list. If the departure arrived before the event of list_events then the actual event on the simulation is the departure, else is the event on list_events.



```python
if(event[0] == "A"):
    # ARRIVAL
    if num_elem_queue < K_num:
        # QUEUE NOT FULL
        n_arrivals += 1
        # Generate service time
        arrival_time = event[1]
        length_packet = self.__generate_exp_distribution(1/avg_len)
        service_time = length_packet/trans_rate

        if num_elem_queue == 0:
            # QUEUE EMPTY
            departure_time = arrival_time + service_time
        elif num_elem_queue < K_num:
            # QUEUE WITH ELEMENTS
            departure_time = last_departure + service_time

        # Adds the new departure time
        departure_list.append(["D", departure_time])
        # Reset the last departure time
        last_departure = departure_time
        num_elem_queue += 1
    else:
        # QUEUE NOT FULL
        lost_packets += 1
    i+= 1
```

If the actual event on the simulation is an arrival (“A”), we first check if the queue is not full, if it is full then we lose a packet and we increase i+=1 meaning that we “processed” a packet of the event list.
If the queue is not full then we update the counter of arrivals, then we calculate the service time of this packet. After generating the service time that if if the queue is empty the departure of this packet is arrival + service time, if the number of elements in the queue is less than K, then the departure of this arrival is the previous departure + service time and then we create the departure event and add it to departure_list and update the respective counters.



```
            elif(event[0] == "O"):
                # OBSERVERS
                n_observers += 1
                total_packs_queue += (n_arrivals - n_departures)
                i+= 1

            elif(event[0] == "D"):
                # DEPARTURE
                n_departures += 1
                num_elem_queue -= 1

    return total_packs_queue/n_observers, lost_packets/n_arrivals
```

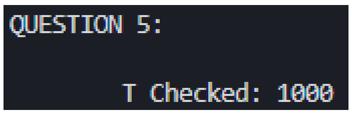
If the event is an observer we update the counters and we register the total packets that are in the queue on the observation time and we processed a packet (i+1)

The last type of event if is a departure we update its counters

And finally we return the observation calculations that in pos 0 is the average E[n] and in pos 1 is the average number of packets lost

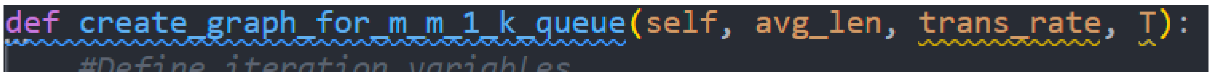
Which simulation T do we take:

To know the simulation Time of the mm1k queue we will run a python program that tells us which simulation time is the better one (with at least a difference of 5% between n*T and (n+1)T. We have done as well as with the infinite queue but with the E[N] and the packet loss



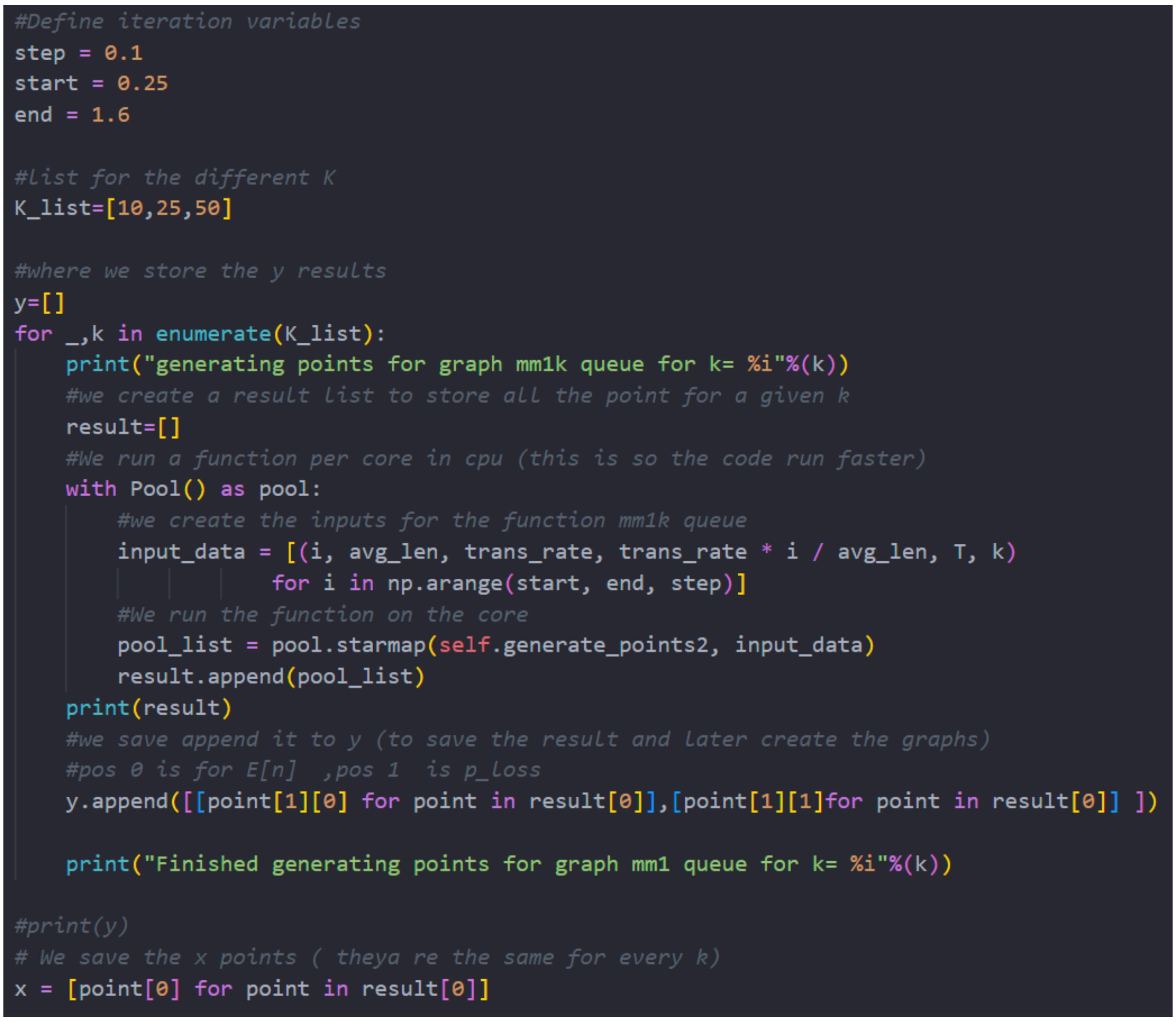
```
QUESTION 5:

        T Checked: 1000
```

QUESTION 6



```
def create_graph_for_m_m_1_k_queue(self, avg_len, trans_rate, T):
    #Define iteration variables
```

To generate the graph we created a python function called create_graph for m m 1 k queue,and we need 3 inputs that are avg_len where we put the average length of a packet and for this graph we used 2000 bits, the trans_rate is the transmission rate represent the bandwidth that our simulation runs and we used 1Mbps (1_000_000) and T is the simulation time that we run the m m 1 k queue.



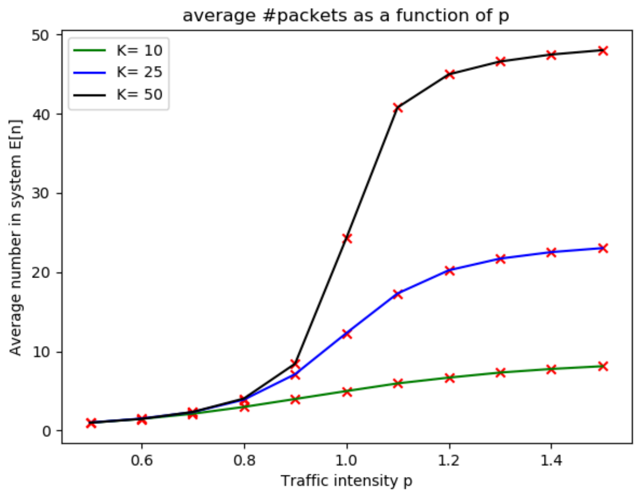
```
#Define iteration variables
step = 0.1
start = 0.25
end = 1.6

#list for the different K
K_list=[10,25,50]

#where we store the y results
y=[]
for _,k in enumerate(K_list):
    print("generating points for graph mm1k queue for k= %i"%(k))
    #we create a result list to store all the point for a given k
    result=[]
    #We run a function per core in cpu (this is so the code run faster)
    with Pool() as pool:
        #we create the inputs for the function mm1k queue
        input_data = [(i, avg_len, trans_rate, trans_rate * i / avg_len, T, k)
                    for i in np.arange(start, end, step)]
        #We run the function on the core
        pool_list = pool.starmap(self.generate_points2, input_data)
        result.append(pool_list)
    print(result)
    #we save append it to y (to save the result and later create the graphs)
    #pos 0 is for E[n]  ,pos 1  is p_loss
    y.append([[point[1][0] for point in result[0]],[point[1][1]for point in result[0]] ])

    print("Finished generating points for graph mm1 queue for k= %i"%(k))

#print(y)
# We save the x points ( theya re the same for every k)
x = [point[0] for point in result[0]]
```

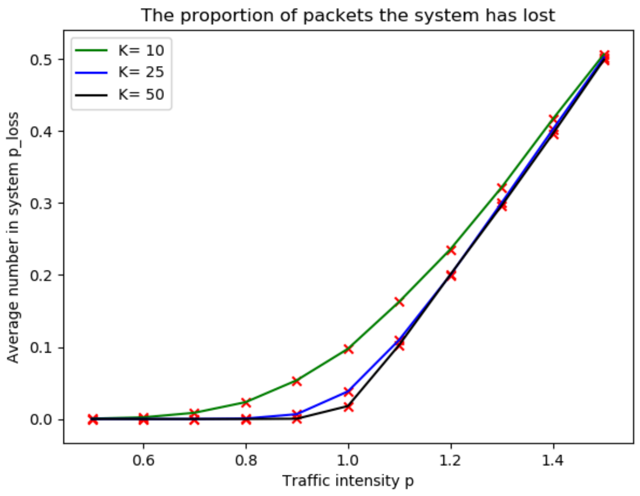
Then inside the function, we first define some iteration variables that are in charge of generating points for 0.5<p<1.5 with steps of 0.1. Then we declared the K_list where we store the max number of packets a queue can hold. After, we declare the list y, where in the pos 0 we will find all y_coordinates for E[n] and p_loss for k=10, in pos 1 all y coordinates for k=25, and in pos 2 all y coordinates for k=50.

Then, we do a loop that we will iterate as K elements we have, in this case 3, we declare the list result where we will store all coordinates (both x and y) for a given y. Then inside the loop we use a library called Pool from multiprocessing. What it does is that we give a list of the different inputs and the function that we want to run independently on different cores of the CPU. The reason that we use this is that to generate these graphs it took a lot of time, and when we used it we saved 80% of time that would have taken without it, and we could correct the errors much faster (parallel programming). After generating all the points for a given K we append to the y_list where the element that we append on pos 0 is E[n] y coordinates, pos 1 the p_loss y coordinates.

After iterating the len(k_list) times, we use the library matplotlib.pylot for generating the graphs and save it on the common folder as exercise_6.png

Results:





As we can observe when k is smaller, the p_loss will be higher earlier than the rest of k as we can see in the right graph, and as we increase the traffic intensity we will lose more packets each time.

According to the E[N], we can see that the average of packets will be higher according to the size of the queue, as well as we increase the traffic intensity.

<u>K=10</u>

p	0.5	0.6	0.7	0.8	0.9	1	1.1	1.2	1.3	1.4	1.5
E[N]	0.989	1.467	2.115	2.978	3.978	4.980	5.951	6.678	7.316	7.773	8.117
Ploss	0.001	0.002	0.009	0.023	0.054	0.098	0.163	0.236	0.322	0.417	0.507

<u>K=25</u>

p	0.5	0.6	0.7	0.8	0.9	1	1.1	1.2	1.3	1.4	1.5
E[N]	1.003	1.506	2.319	3.897	7.107	12.291	17.303	20.228	21.7	22.52	23.026

Ploss	0	0	0	0.001	0.007	0.038	0.11	0.2	0.3	0.402	0.502

<u>K=50</u>

p	0.5	0.6	0.7	0.8	0.9	1	1.1	1.2	1.3	1.4	1.5
E[N]	0.988	1.47	2.34	4.033	8.433	24.278	40.782	44.955	46.577	47.458	48.008
Ploss	0	0	0	0	0	0.018	0.103	0.201	0.296	0.396	0.499

Appendix

```
"""
University of Waterloo Fall 2023 ECE-358 LAB-1  Group 151
József IVÁN GAFO (21111635) jivangaf@uwaterloo.ca
Sonia NAVAS RUTETE (21111397) srutete@uwaterloo.ca
V 1:0
In this module we will write the main code
"""

# Imports
from multiprocessing import Pool
import matplotlib.pyplot as plt
import numpy as np
import math
import random
import os


# We define the type of events
ARRIVAL = "A"
DEPARTURE = "D"
OBSERVER = "O"


class Lab1():
    """
    This class is in charge of containing all the
    methods required for the lab 1
    """

    def __init__(self) -> None:
        pass

    # We write the main code
    def question1(self, lambda_param):
        """
        This method is in charge of generating 1000 random variables
        with lambda 75 with numpy library and it prints the mean and variance
        @param lambda_param: Value for lambda
        @return : None
        """
        # For this exercise we will use the library numpy
        # as it makes the operations easier

        # Calculate the expected mean and variance for an exponential
distribution with λ=75
        expected_mean = 1 / lambda_param
        expected_variance = 1 / (lambda_param ** 2)

        # We generate 1000 exponential random variables
        generated_numbers = [self.__generate_exp_distribution(
            lambda_param) for _ in range(1000)]

        # We do the mean and the variance
        mean = np.mean(generated_numbers)
        variance = np.var(generated_numbers)

        # Print the results
        print("\nQUESTION 1:")
        print(f"\tMean of generated random variables: {mean}")
        print(f"\tExpected mean for λ=75: {expected_mean}")
        print(f"\tVariance of generated random variables: {variance}")
        print(f"\tExpected variance for λ=75: {expected_variance}\n")

    def m_m_1_queue(self, avg_len: int, trans_rate: int, lambda_par: int, T: int)
-> list:
        """
        Build your simulator for this queue and explain in words what you have
done. Show your code in the report. In
        particular, define your variables. Should there be a need, draw diagrams
to show your program structure. Explain how you
        compute the performance metrics. Type of queue M/M/ (infinite queue)
        @param avg_len: Is the average length of a packet in bits (L)
        @param trans_rate: Is the transmission rate of the output link in
bits/sec (C)
        @return: None
        """
        # ! Declaration of variables for the m_m_1 queue
        num_arrival = 0
        num_departed = 0
        num_observers = 0
        transmission_times = []
        arrival_list = []
        departure_list = []
        observer_list = []
        event_list = []
        result_list = []

        # * Arrival

        # we generate the arrivals
        arrival_list = self.__generate_mm1_arr_obs(lambda_par, T)


        # * observers
        # Now we add the observers event
        observer_list = self.__generate_mm1_arr_obs(lambda_par*5, T)






        # * Departure
        # * generate packet lengths for each arrival
        length_packets = []
        # We create the packet size for each arrival
        length_arrival = len(arrival_list)
        length_packets = [self.__generate_exp_distribution(1/avg_len) for _ in
range(length_arrival)]

        # * Calculate how much time takes to process all the packet

        for packet in length_packets:
            transmission_times.append(packet / trans_rate)

        # *We calculate the departure time
        queue_time = 0
        departure_time = 0
        # Loop that calculates how the time of departure for each packet
        for count, arrival_packet_time in enumerate(arrival_list):
            # If the queue is idle we just sum the arrival time +transmission
[count] and we add it to the list
            if queue_time < arrival_packet_time:
                departure_time = arrival_packet_time+transmission_times[count]
            # Else there is a queue, and we add the last package  departure time
(count-1)  + the transmission[count]
            else:
                departure_time = departure_list[count-1] +
transmission_times[count]
            departure_list.append(departure_time)
            # We update the queue time of the queue, with the las departure time
            queue_time = departure_time


        # * Order all packets by time with its type
        for arrival_time in arrival_list:
            result_list.append(["A", arrival_time])
        for departure_time in departure_list:
            result_list.append(["D", departure_time])
        for observer_time in observer_list:
            result_list.append(["O", observer_time])
        # We sort all the time events by time
        event_list = sorted(result_list, key=lambda x: x[1])



        # * We calculate E[n] and p_idle
        # Declaration of variables
        total_num_packs_queue = 0
        total_observer_idles = 0

        for _, event in enumerate(event_list):

            event_type = event[0]
            # Arrival
            if event_type == 'A':
                num_arrival += 1
            elif event_type == 'D':
                num_departed += 1
            else:
                num_observers += 1
                # We record the num of packets that are currently on the queue
                total_num_packs_queue += (num_arrival-num_departed)
                if num_arrival == num_departed:
                    total_observer_idles += 1
        return total_num_packs_queue/num_observers,
total_observer_idles/num_observers

    def __generate_mm1_arr_obs(self, lambda_par,T):
        """
        This method is in charge of generating the list of arrivals and observers
        @param lambda_param: An integer that contains the average number of
packets arrived
        @param T: Duration of the simulation
        @param steps: It is 1 for default, will change for observers as the param
is different
        @return list: We return a list with the events generated
        """
        # Iteration variables
        aux_list = []
        simulation_time = 0
        #Loop we iterate until we reach the simulation time
        while simulation_time < T:
            #We generate an exponential distribution
            arrival_timestamp =
self.__generate_exp_distribution(lambda_par)+simulation_time
            #We add to the aux list
            aux_list.append(arrival_timestamp)
            #We update the simulation time with the new arrival
            simulation_time = arrival_timestamp
        #We return the aux list
        return aux_list

    def __generate_exp_distribution(self, lambda_param: int) -> list:
        """
        This method is in charge of generating exponential random variables
        @param lambda_param: An integer that contains the average number of
packets arrived
        @param size: An integer that defines how many numbers we generate
        @return list: We return a list with the numbers generated
        """
        expected_mean = 1/lambda_param
        return -expected_mean*math.log(1-random.random())

    def m_m_1_k_queue(self, avg_len:int,
trans_rate:int,lambda_par:int,T:int,K_num:int)->[float,float,float]:
        """
        This method is in charge of simulating the m_m_1_k queue
        @param avg_len: This integer represent the average length packet in bits
        @param trans_rate: This integer represent the transmission rate of a
packet.
        @param lambda_par: This integer represents the parameter lambda od the
poisson distribution
        @param T: This integer represent the simulation time
        @param K: This integer represent the max number of packets that a queue
can hold
        @return a list: It returns a list of floats where the first element
represent E[n],p_idle and p_loss
        """
        #We declare variables
        num_elem_queue = 0
        n_arrivals = 0
        n_observers = 0
        n_departures = 0

        total_packs_queue = 0
        lost_packets = 0

        last_departure = 0
        departure_list = []



        # Generating arrivals
        list_arrivals = []
        list_arrivals = self.__generate_mm1_arr_obs(lambda_par, T)

        #Generating the observers
        list_observers = []
        list_observers  = self.__generate_mm1_arr_obs(lambda_par*5, T)

        #We add event on event_list  where in pos 0 we define the type
("A"=arrival, "O"=observers)
        #in pos 1 we have the time_stamp of the event
        list_events = []
        for e in list_arrivals:
            list_events.append(["A",e])

        for e in list_observers:
            list_events.append(["O",e])

        #we sort the evnt list by event arrival time
        list_events = sorted(list_events, key=lambda x: x[1])



        #Loop where we will calculate the departure of the arrivals
        #calculate the observers stadistics
        i = 0
        while i < len(list_events) or departure_list != []:
            if(i == len(list_events)):
                # If list_events has finished but we still have departures
                for x in range(len(departure_list)):
                    departure_list.pop(0)
                    n_departures += 1
                    num_elem_queue -= 1
            else:
                # Assign current event
                if(departure_list != []):
                    # If there are departures
                    if(departure_list[0][1] < list_events[i][1]):
                        # Check if it goes an observer, arrival or departure
                        event = departure_list.pop(0)
                    else:
                        event = list_events[i]
                else:
                    event = list_events[i]

                if(event[0] == "A"):
                    # ARRIVAL
                    if num_elem_queue < K_num:
                        # QUEUE NOT FULL
                        n_arrivals += 1
                        # Generate service time
                        arrival_time = event[1]
                        length_packet = 
self.__generate_exp_distribution(1/avg_len)
                        service_time = length_packet/trans_rate

                        if num_elem_queue == 0:
                            # QUEUE EMPTY
                            departure_time = arrival_time + service_time
                        elif num_elem_queue < K_num:
                            # QUEUE WITH ELEMENTS
                            departure_time = last_departure + service_time

                        # Adds the new departure time
                        departure_list.append(["D", departure_time])
                        # Reset the last departure time
                        last_departure = departure_time
                        num_elem_queue += 1
                    else:
                        # QUEUE NOT FULL
                        lost_packets += 1
                    i+= 1


                elif(event[0] == "O"):
                    # OBSERVERS
                    n_observers += 1
                    total_packs_queue += (n_arrivals - n_departures)
                    i+= 1

                elif(event[0] == "D"):
                    # DEPARTURE
                    n_departures += 1
                    num_elem_queue -= 1

        return total_packs_queue/n_observers, lost_packets/n_arrivals



    #Generate graphs

    def generate_point(self, i, avg_len, trans_rate, lambda_par, T):
        # Calculate data point for a specific 'i'
        list_m_m_1 = self.m_m_1_queue(avg_len, trans_rate, lambda_par, T)
        # If we want E[n] then type_info is 0 if is p_idle then type_info is 1
        return [i, list_m_m_1]

    def create_graph_for_m_m_1_queue(self, avg_len, trans_rate, T):
        step = 0.1
        start = 0.25
```

```
        end = 1.05
        # Graph for E[N]
        result=[]
        print("Generating points for graph mm1 queue:")
        #cores=4
        with Pool() as pool:
            input_data = [(i, avg_len, trans_rate, trans_rate * i / avg_len, T)
                        for i in np.arange(start, end, step)]
            pool_list = pool.starmap(self.generate_point, input_data)
            print(pool_list)
        print("\nFinished generating points for graph mm1 queue.\n")
        result.append(pool_list)


        # We save the points
        #pos 0 is for E[n] and pos 1 is for p_idle
        x = [point[0] for point in result[0]]
        y = [[point[1][0] for point in result[0]],
        [point[1][1] for point in result[0]]]

        #We create the graph
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
        graph_list=[ax1,ax2]
        text=[['Average number in system E[n]',"average #packets as a function of
p"],
              ["Average number in system p_idle","The proportion of time the
system is idle as a function of p"]]
        for i,graph in enumerate(graph_list):
            # We initialize the graph
            graph.scatter(x, y[i],color="red",marker='x')
            graph.plot(x, y[i],label="K is infinite")
            graph.set_xlabel('Traffic intensity p')
            graph.set_ylabel(text[i][0])
            graph.set_title(text[i][1])
            graph.legend()
        #We save it
        # Save the figure as an image in the "graphs" folder
        script_directory = os.path.dirname(__file__)

        # Save the figure as an image named "exercise_3.png" in the same folder
as the script
        image_path = os.path.join(script_directory, 'exercise_3.png')
        plt.savefig(image_path)
        plt.close()

    def create_graph_for_m_m_1_k_queue(self, avg_len, trans_rate, T):
        #Define iteration variables
        step = 0.1
        start = 0.5
        end = 1.6

        #list for the different K
        K_list=[10,25,50]

        #where we store the y results
        y=[]
        for _,k in enumerate(K_list):
            print("Generating points for graph mm1k queue for k= %i\n"%(k))
            #we create a result list to store all the point for a given k
            result=[]
            #We run a function per core in cpu (this is so the code run faster)
            with Pool() as pool:
                #we create the inputs for the function mm1k queue
                input_data = [(i, avg_len, trans_rate, trans_rate * i / avg_len,
T, k)
                                for i in np.arange(start, end, step)]
                #We run the function on the core
                pool_list = pool.starmap(self.generate_points2, input_data)
                result.append(pool_list)
            #we save append it to y (to save the result and later create the
graphs)
            #pos 0 is for E[n]  ,pos 1  is p_loss
            print(result)
            y.append([[point[1][0] for point in result[0]],[point[1][1]for point
in result[0]] ])

            print("\nFinished generating points for graph mm1 queue for k=
%i\n"%(k))

        # We save the x points ( theya re the same for every k)
        x = [point[0] for point in result[0]]



        #We create the graph
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
        graph_list=[ax1,ax2]
        #variables for creating the graph
        text=[['Average number in system E[n]',"average #packets as a function of
p"],
              ["Average number in system p_loss", "The proportion of packets the
system has lost"]]
        colours=["green","blue","black"]
        for i,graph in enumerate(graph_list):
            # We initialize the graph
            for k,k_number in enumerate(K_list):
                label_name= "K= %i"%(k_number)
                graph.scatter(x, y[k][i],color="red",marker='x')
                graph.plot(x, y[k][i],label=label_name,color=colours[k])
            graph.set_xlabel('Traffic intensity p')
            graph.set_ylabel(text[i][0])
            graph.set_title(text[i][1])
            #we write the legend
            graph.legend()
        #We save it
        # Save the figure as an image in the "graphs" folder
        script_directory = os.path.dirname(__file__)

        # Save the figure as an image named "exercise_3.png" in the same folder
as the script
        image_path = os.path.join(script_directory, 'exercise_6.png')
        plt.savefig(image_path)
        plt.close()



    def generate_points2(self,i,avg_len,trans_rate,lambda_par,T,K):
        # Calculate data point for a specific 'i'
        list_m_m_1 = self.m_m_1_k_queue(avg_len, trans_rate, lambda_par, T,K)
        # If we want E[n] then type_info is 0 if is p_idle then type_info is 1
        return [i, list_m_m_1]

def check_T(avg_len, trans_rate, lambda_par, T):
    a = Lab1()
    T_counter = 1
    percentage = 0.05
    dif_count_E = 100
    dif_count_pidle = 100
    list_T = []
    gate = True
    while gate:
        E, pidle = a.m_m_1_queue(avg_len, trans_rate, lambda_par, T_counter*T)
        E2, pidle2 = a.m_m_1_queue(avg_len, trans_rate, lambda_par,
(T_counter+1)*T)
        difference_E = abs(E-E2)
        difference_pidle = abs(pidle-pidle2)
        if(len(list_T) == 2):
            return min(list_T[0], list_T[1]) * T
        if(difference_E <= E*percentage and difference_pidle <=
pidle*percentage):
            if dif_count_E > difference_E and dif_count_pidle > difference_pidle:
                list_T.append(T_counter+1)
                dif_count_E = difference_E
                dif_count_pidle = difference_pidle
        T_counter += 1

def check_T2(avg_len, trans_rate, lambda_par, T, K):
    a = Lab1()
    T_counter = 1
    percentage = 0.05
    dif_count_E = 100
    dif_count_ploss = 100
    list_T = []
    gate = True
    while gate:
        E, ploss = a.m_m_1_k_queue(avg_len, trans_rate, lambda_par, T_counter*T,
K)
        E2, ploss2 = a.m_m_1_k_queue(avg_len, trans_rate, lambda_par,
(T_counter+1)*T, K)
        difference_E = abs(E-E2)
        difference_ploss = abs(ploss-ploss2)
        if(len(list_T) == 2):
            return min(list_T[0], list_T[1]) * T
        if(difference_E <= E*percentage and difference_ploss <=
ploss*percentage):
            if dif_count_E > difference_E and dif_count_ploss > difference_ploss:
                list_T.append(T_counter+1)
                dif_count_E = difference_E
                dif_count_ploss = difference_ploss
            else:
                gate = False
        else:
            gate = False
        T_counter += 1
    return T

if __name__ == "__main__":
    a = Lab1()
    lambda_par = 75
    trans_rate = 1_000_000
    avg_packet_length = 2_000
    T = 1_000

    # RUNNING THE LAB
    # QUESTION 1
    a.question1(lambda_par)

    # INFINITE QUEUE
    # QUESTION 2
    print("QUESTION 2:")
    X = check_T(avg_packet_length, trans_rate, lambda_par, T)
    print("\t T Checked : " + str(X))

    # QUESTION 3
    print("QUESTION3:\n")
    print("The graph will be generated in exercise3.png\n")
    a.create_graph_for_m_m_1_queue(avg_packet_length,trans_rate,2*T)


    # QUESTION 4
    # For p=1.2
    print("QUESTION 4:")
    E, pidle = a.m_m_1_queue(avg_packet_length, trans_rate, trans_rate * 1.2 /
avg_packet_length, 2*T)
    print("\tFor p = 1.2, the value of E[N] = "+ str(E) + " and the value of
pidle =" + str(pidle))


    # FINITE QUEUE
    # QUESTION 5
    print("QUESTION 5:")
    trans_rate = 1_000_000
    avg_packet_length = 2_000
    T = 1000
    k = [10, 25, 50]
    X = check_T2(avg_packet_length, trans_rate, trans_rate * 0.5 /
avg_packet_length, T, 10)
    print("\tT Checked: " + str(X))

    #Question 6
    print ("QUESTION 6:\n")
    print("The graph will be generated in exercise_6.png \n")
    b = Lab1()
    b.create_graph_for_m_m_1_k_queue(avg_packet_length,trans_rate,X)
```