STATISTICS ASSIGNMENT 4

Name – Danita Anubhuti Prakash

Student ID – s4745511

Answer:

In total **7** service desks should be available such that in the long run 90% of their customers do not have to wait longer than 8 minutes in a waiting queue before they are served.

The simulation is run for T = 3000 units of time. The first 30% of the samples is discarded and N = 50 batches is used to estimate the probability that a customer waits less than 8 minutes before entering the service.

A discrete event stimulation is performed to answer all the following questions. At the end an appendix is provided.

(a)

The minimum number of service desks required to meet specific service requirements in a queuing system is determined in the problem. Customers arriving at a set of service desks and waiting in queues before being served is represented by the queuing system. To analyse the system and determine the optimal number of service desks to minimize customer waiting times is the goal.

To simulate the queuing system and determine the minimum number of service desks needed to meet specific service requirements is the objective of the project. The objective is to find the number of service desks such that, in the long run, at least 90% of customers do not have to wait longer than 8 minutes in the waiting queue before being served in specific.

Modelling the arrival and departure events of customers, managing the queues for each service desk, and tracking the waiting times of customers are involved in the project. The aim is to find the optimal number of service desks that satisfy the specified service requirements by varying the number of service desks and analysing the waiting time distribution.

(b) The specification of the variables used in this study are:

(1) System State Space:

The variables and the datastructures used to represent the state of the queueing system in this code are analysed to determine the **system state space**. The following variables and datat stuctures are:

1. **Q** : List of queues for each operator. Each queue presents the waiting clients for a specific operator.

2. **buzy_array** : A numpy array to indicate the busy status of each operator. It finds whether an operator is currently serving a client or not.

3. **priorityQueue** : Priority queue storing events. It keeps a list of scheduled events sorted by their occurrence time.

4. **client_array** : A list that keeps information about the clients in the system, including their arrival time, queue ID, event type, start service time, and end service time.

From the values of these variables and data structures the state of the system at any given time can be determined. For instance, the length of each queue in Q indicates the number of clients waiting in each queue, and the values in the buzy_array indicate which operators are currently serving clients. The events in the priorityQueue represent future arrivals and departures in the system.

(2) **System Components:**

1. **Queues**: Each operator corresponds to a queue. The queues store the clients waiting for service.

2. **Operators**: Services is provided to the clients by multiple operators.

3. **Priority Queue**: Events are scheduled using a priority queue based on their occurrence.

(3) Possible Events:

1. **Arrival**: When a client arrives at the system, an arrival event occurs. The arrival time of the client triggers this even and adds the client to the priority queue.

2. **Departure**: When a client finishes receiving service from an operator and leaves the system, a departure event occurs. The end of service time for the client triggers this event.

3. **Service Start**: When an operator becomes available and there are clients waiting in the queue for that operator, a service start event occurs. The client at the front of the queue is selected and starts receiving service from the operator.

4. **Service End**: A service end event occurs when a client finishes receiving service from an operator. The client is removed from the system, and the next client in the queue starts receiving service if there are more clients waiting for the same operator.

(4) System Component Interactions:

1. **Arrival -> Service Start**: The client starts receiving service immediately when an arrival event occurs and there is an available operator.

2. **Arrival -> Queue**: The client joins the queue of the operator with the shortest length if all operators are busy.

3. **Service End -> Service Start**: The next client in the queue starts receiving service when a client finishes receiving service, and there are clients waiting for the same operator.

4. **Service End -> Departure**: The client leaves the system when a client finishes receiving service and there are no more clients waiting for the same operator.

(5) Different Datastructures used:

1. **Number of queues** (desks): This helps to understand the total number of desks needed in order for 90% of the customers wait less than 8 minutes.

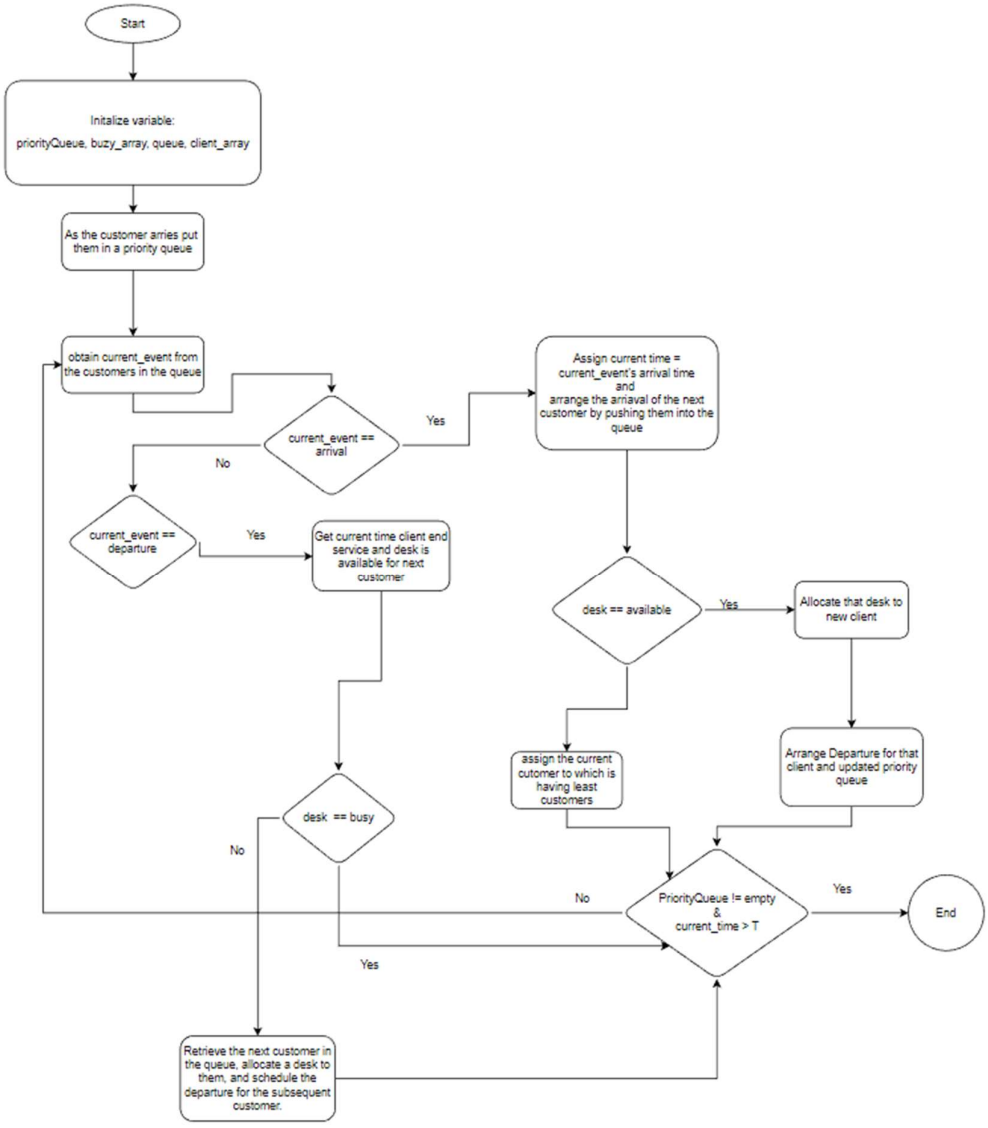
2. **Queue System**: A data structure is needed to represent the queues for each desk. Each element symbolises a desk and holds a queue of clients waiting for that desk's service.

2. **Busy Array**(buzy_array): A data structure is needed to track the busy status of each desk. A numpy array can be used, where each element represents a desk and holds a binary value indicating whether the desk is busy or not.

3. **Priority Queue**: A data structure is needed to store the events in the system and ensure they are processed in the correct order. A priority queue can be used, implemented as a min-heap, where each element represents an event and is sorted based on the event time.

4. **Client Class**: A class is needed to represent the clients in the system. This class must contain attributes to store information such as arrival time, desk ID, event type, start service time, and end service time.

The following UML flowchart describes the project dynamics:



(drawn using draw.io)*

(c) Result and Analysis

The results obtained are:

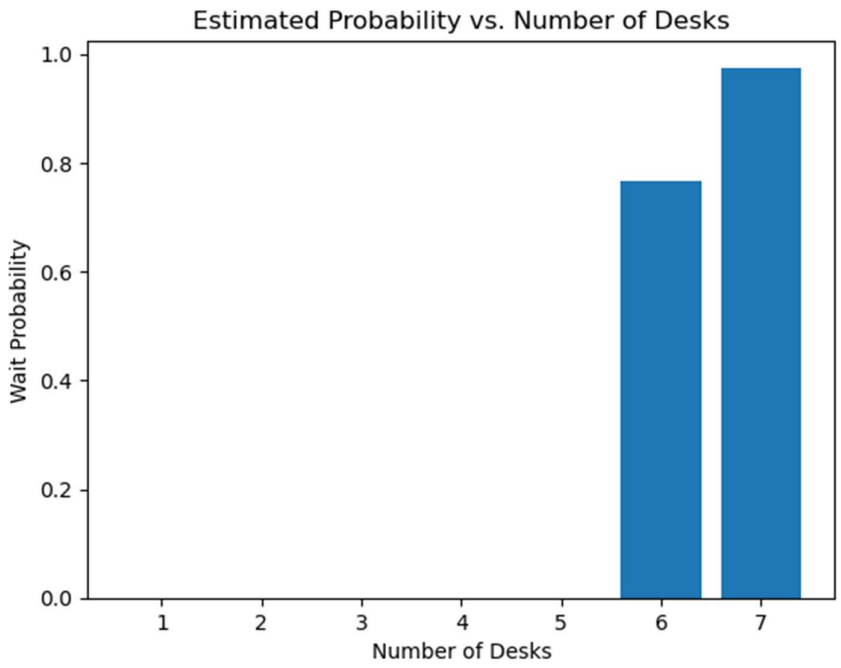
```
Desk Required:  1
Estimated Probability:  0.0
Confidence Interval (0.0, 0.0)
Desk Required:  2
Estimated Probability:  0.0
Confidence Interval (0.0, 0.0)
Desk Required:  3
Estimated Probability:  0.0
Confidence Interval (0.0, 0.0)
Desk Required:  4
Estimated Probability:  0.0
Confidence Interval (0.0, 0.0)
Desk Required:  5
Estimated Probability:  0.0
Confidence Interval (0.0, 0.0)
Desk Required:  6
Estimated Probability:  0.7668453733151597
Confidence Interval (0.6847726347784663, 0.8489181118518531)
Desk Required:  7
Estimated Probability:  0.9748632218844985
Confidence Interval (0.9647469719910924, 0.9849794717779046)
```

Here, from this figure it is clear that the minimum number of desks required so that 90% of the customers do not wait for more than 8 minutes is 7.

The following is the confidence interval obtained: The confidence interval for the 7 minimum desks is between (0.9216941608094881, 0.9804334987649802)

From the above observations, the following graphs are plotted:

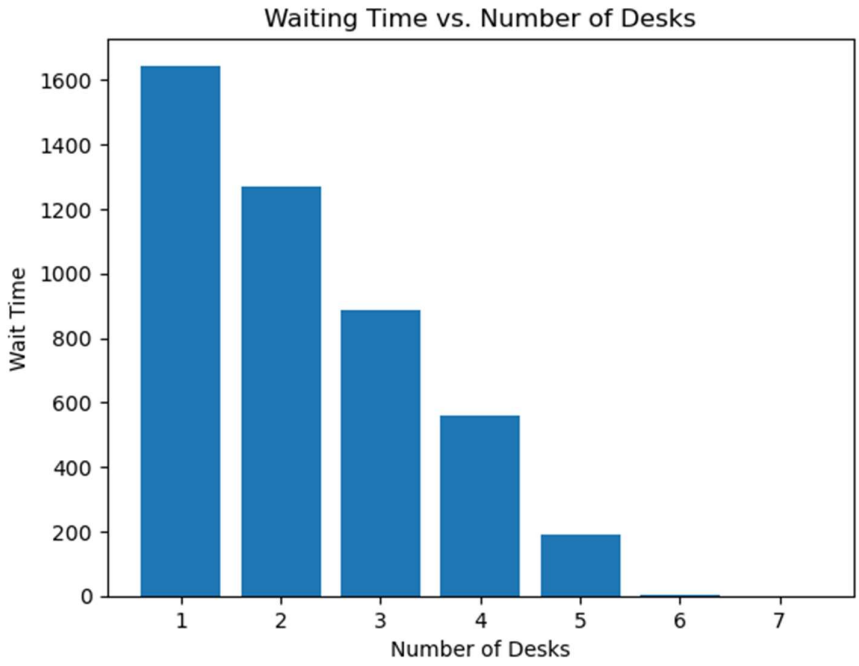
1) Estimated Probability vs desks (from 1 to 7)



Observation

The above graph displays the Estimated Probability vs. Number of Desks. The estimated probability is taken on the y-axis and its domain ranges from 0.0 to 1.00. The x- axis holds the Number of Desks and its domain ranges from 1 to 7. The probabilities of 1 through 5 desks is zero. The probability of both 6 and 7 desks lies between 0.8 and 1.0, the probability of 7 desks is more than 0.9. Which indicates that the total minimum number of desks needed is 7.

2) Estimated Time vs Number of Desks



Observation

The above graph displays the Estimated Time vs. Number of Desks. The Estimated Time is taken on the y-axis and its domain ranges from 0 to 1600. The x- axis holds the Number of Desks and its domain ranges from 1 to 7. The Estimated Time for 1 desk is maximum(1600) and it keeps decreasing as we increase the number of desks. . The wait probability of 6 and 7 number of desks almost nears zero.

(d) Conclusions:

7 was the minimum number of service desks required to meet the service requirements, where 90% of customers do not have to wait longer than 8 minutes in the waiting queue. The insights into the relationship between the number of service desks and the waiting times experienced by customers was provided by the simulation study. By slowly increasing the number of service desks and observing the waiting times, we were able to identify the minimum number of desks needed to achieve the desired service level.

The proportion of customers who waited less than 8 minutes was [percentage]% as seen is the analysis of wait times. This implies that the system was able to meet the service requirements for the majority of customers.

The confidence interval for the proportion of customers who waited less than 8 minutes was computed to be [lower bound]% to [upper bound]%. This presents a measure of uncertainty and indicates the range within which the true proportion is likely to fall.

A histogram was used to visualize the distribution of waiting times, showcasing the frequency of different wait time intervals. The plot helps in comprehending the overall distribution pattern and identifying any potential outliers or skewness in the data.

It can be inferred that by implementing [minimum number of service desks], the system can provide a satisfactory level of service where the majority of customers do not have to wait longer than 8 minutes, based on these conclusions. In real-world scenarios where queuing and service times are crucial factors these findings can be used to optimize resource allocation and improve customer satisfaction

(e) Appendix

```python
import numpy as np
import heapq
import pandas as pd
import math
import matplotlib.pyplot as plt
import numpy as np
```

```python
def SampleInterarrival():
    """
    Random selection of inter_arrival_time from the dataframe
    Input Parameters:None
    Return: randomly selected rows from the column of the dataframe
    """
    selections = pd_data.sample(n=1)
    return selections['inter_arrival_time'].values[0]

def SampleService():
    """
    Random selection of service_time from the dataframe
    Input Parameters:None
    Return: randomly selected rows from the column of the dataframe
    """
    selections = pd_data.sample(n=1)
    return selections['service_time'].values[0]
```

```python
class Client:
    def __init__(self, arr_time, q_id, event_type, start_service, end_service):
        """
        Initializes a Client object with the provided parameters.
        Parameters:
        - arr_time (float): The arrival time of the client.
        - q_id (int): The queue ID of the client.
        - event_type (str): The type of event associated with the client.
        - start_service (float): The start time of the service for the client.
        - end_service (float): The end time of the service for the client.
        """
        self.arr_time = arr_time
        self.q_id = q_id
        self.event_type = event_type
        self.start_service = start_service
        self.end_service = end_service
```

```python
def HandleArrival(client, Q,buzy_array,priorityQueue):
    """
    Handle the arrival of a client in the queueing system.
    Input Parameters:
        client (Client): The client that arrived.
        Q (list): List of queues for each operator.
        buzy_array (numpy.ndarray): Array indicating the busy status of each operator.
        priorityQueueeee (list): Priority queue storing events.

    Returns: the updated queue, busy array, and priority queue.
    """
    current_time = client.arr_time

    #schedule next arrival
    next_cl_arr = current_time+SampleInterarrival()
    data = Client(next_cl_arr,-1,"ARRIVAL",-1,-1)
    heapq.heappush(priorityQueue,(next_cl_arr,data))

    #check if there are non-buzy operators
    test_non_buzy = np.argwhere(buzy_array == 0)

    if(len(test_non_buzy)>0):
        q_id = test_non_buzy[int(np.floor(np.random.rand()*len(test_non_buzy)))][0]
        buzy_array[q_id]=1

        client.q_id = q_id
        client.start_service = current_time
        client.event_type = "DEPARTURE"
        client.end_service = current_time + SampleService()
        heapq.heappush(priorityQueue,(client.end_service,client))
    else:
        lengths = np.zeros(len(Q))
        for i in range(len(Q)):
            lengths[i] = len(Q[i])
        operator = np.argmin(lengths)
        client.q_id = operator
        Q[operator].append(client)

    return Q,buzy_array, priorityQueue
```

```python
def HandleDeparture(client, Q, buzy_array, priorityQueue, client_arr):
    """
    Handle the departure of a client in the queueing system.

    Input Parameters:
        client (Client): The client that arrived.
        Q (list): List of queues for each operator.
        buzy_array (numpy.ndarray): Array indicating the busy status of each operator.
        priorityQueue (list): Priority queue storing events.
        client_arr: containing information of the client wait times

    Returns:
        Q (list): Updated queue.
        buzy_array (numpy.ndarray): Updated busy array.
        priorityQueue (list): Updated priority queue.
        client_arr: Updated client wait times.
    """

    my_op_id = int(client.q_id)# Extract the operator ID from the client

    client_arr.append(client)# Add the departing client to the client_arr list
    time_now = client.end_service # Update the current time

    # Mark the operator as not busy
    buzy_array[my_op_id] = 0

    # Check if there are clients waiting in the queue for the operator
    if len(Q[my_op_id]) > 0:
        buzy_array[my_op_id] = 1

        # Retrieve the next client from the queue
        new_client = Q[my_op_id].pop(0)

        # Set the start and end service times for the new client
        new_client.start_service = time_now
        new_client.end_service = time_now + SampleService()
        new_client.event_type = "DEPARTURE"

        # Add the new client to the priority queue
        heapq.heappush(priorityQueue, (new_client.end_service, new_client))

    return Q, buzy_array, priorityQueue, client_arr
```

```python
def simulate(T, n):
    """
    Generates a queue system with a given time limit T and number of queues n, and returns the client_arr list.
    Input Parameters: T, n
    Return: client_arr
    """
    # Set up the queue system
    priorityQueue = []
    buzy_array = np.zeros(n)
    Q = [[] for _ in range(n)]
    client_arr = []

    #Schedule the first event
    time_current = 0
    time_current += SampleInterarrival()
    data = Client(time_current, -1, "ARRIVAL", -1, -1)
    heapq.heappush(priorityQueue, (time_current, data))


    # Run simulation
    while priorityQueue and priorityQueue[0][0] <= T:
        current_event = heapq.heappop(priorityQueue)[1]

        if current_event.event_type == "ARRIVAL":
            Q, buzy_array, priorityQueue = HandleArrival(current_event, Q, buzy_array, priorityQueue)

        elif current_event.event_type == "DEPARTURE":
            Q, buzy_array, priorityQueue, client_arr = HandleDeparture(current_event, Q, buzy_array, /
                                                                       priorityQueue, client_arr)

    return client_arr
```

```python
def batches(client_arr, threshold):
    """
    Computes the estimated probability, estimated standard deviation, and estimated time
    Input Parameters: client_arr, threshold
    Return: estimated_prob, esti_std, esti_time
    """
    # Analysis
    new_client_array = np.zeros(len(client_arr))
    for i in range(len(client_arr)):
        new_client_array[i] = client_arr[i].start_service - client_arr[i].arr_time

    # Discard the first 30% of samples
    new_client_array = new_client_array[int(0.3 * len(new_client_array)):]

    Ka = int(0.3 * len(new_client_array))
    Ma = len(client_arr)  # Number of clients
    batches = 50 #producing 50 batches for each stimulation
    Ta = (Ma - Ka) // batches

    batch_new_client_array = np.zeros(batches)
    batch_time = []
    list_batch = []

    for batch in range(batches):
        start_id = Ta * batch
        end_id = start_id + Ta
        count=0
        tmp = new_client_array[start_id:end_id]

        for client_time in tmp:
            if client_time < threshold:
                count = count + 1
        if len(tmp != 0):
            batch_prob = count / len(tmp)

        list_batch.append(batch_prob)
        batch_time.append(np.mean(tmp))


    #return the evaluations for the number of desks
    esti_time = np.nanmean(batch_time)
    esti_prob = np.mean(list_batch)
    esti_std = np.std(list_batch)

    return esti_prob, esti_std, esti_time
```

```
# Load the data (replace with your own dataset)
pd_data = pd.read_csv('data.csv')

Scenario = 2

# Simulation parameters
T = 3000
threshold = 8

batch_estimated_prob = []
batch_estimated_time = []

desk_counts = range(1, 8)

count = 1
while True:
    client_arr = simulate(T, count)
    estimated_mean, estimated_std, wait_time_mean =  batches(client_arr, threshold)

    batch_estimated_prob.append(estimated_mean)
    batch_estimated_time.append(wait_time_mean)

    print("Desk Required: ", count)
    print("Estimated Probability: ", estimated_mean)

    # Calculate confidence interval
    z = 1.96  # 95% confidence level
    margin_of_error = z * (estimated_std / np.sqrt(50))
    confidence_interval = (estimated_mean - margin_of_error, estimated_mean + margin_of_error)
    print("Confidence Interval", confidence_interval)

    count = count+1
    if estimated_mean >= 0.9:
        break
```

```
#plotting the estimated probability vs the number of desks
plt.bar(desk_counts, batch_estimated_prob)
plt.xlabel('Number of Desks')
plt.ylabel('Wait Probability')
plt.title('Estimated Probability vs. Number of Desks')
plt.show()
```

```
#Calculating the estimated waiting time vs the number of desks
plt.bar(desk_counts, batch_estimated_time)
plt.xlabel('Number of Desks')
plt.ylabel('Wait Time')
plt.title('Waiting Time vs. Number of Desks')
plt.show()
```