Homework #2

Name: Hanwen Li **Email:** lihanwen@seas.upenn.edu

In [ ]:
```
import numpy as np
def get_divider(id):
    print('**************************************************** Problem {} ****************************************************'.format(id))
```

Problem #1

The `Problem1` class is designed to simulate a queue system where customers pass through a counter and then proceed to checkout. This simulation is useful for understanding and analyzing customer wait times, service efficiency, and idle times in a service-oriented environment.

Constructor: `__init__(self, num)`

- Initializes the `Problem1` instance with the specified number of customers (`num`).
- Attributes initialized include arrays for checkout, counter, and arrival times, along with dictionaries for interarrival times, counter service times, checkout service times, and dictionaries to store the start and end times of service for each customer at both the counter and checkout.
- Calls methods to generate arrival, counter, and checkout times upon instantiation.

Parameters

- `num` : The total number of customers to simulate.

Attributes

- `checkout_time` , `counter_time` , `arrive_time` : Arrays to store times for each customer at different stages.
- `num` : Total number of customers.
- `interarrival` , `counter` , `checkout` : Dictionaries holding possible times and their probabilities for different stages.
- `time_service_begin` , `time_service_end` : Dictionaries to store service start and end times.

Methods

`get_arrive_time(self, length)`

- Generates and sets arrival times for customers based on specified interarrival times and probabilities.

`get_counter_time(self, length)`

- Generates and sets service times at the counter for all customers.

`get_checkout_time(self, length)`

- Generates and sets service times at checkout for all customers.

`get_wait(self)`

- Calculates waiting and idle times at both counter and checkout stages.

- Returns arrays of waiting times and a dictionary of idle times, providing insights into customer wait times and service point efficiency.

Simulation Results

After creating an instance of `Problem1` with a specified number of simulations (e.g., 5000 customers), the simulation calculates and prints average waiting times, probabilities of waiting, and the fraction of time each server (counter and checkout) is idle. This allows for a comprehensive analysis of the simulated queue system.

Example Output

The code snippet provided at the end demonstrates how to instantiate the class, invoke the `get_wait` method to perform the simulation, and then print out various statistics such as average waiting times, the probability of customers waiting in line, and server idle times.

In [ ]:
```
class Problem1:
    def __init__(self, num):
        """
        Initializes the Problem1 instance with the specified number of customers (num).

        Attributes:
        - checkout_time: Array to store the time each customer spends at checkout.
        - counter_time: Array to store the time each customer spends at the counter.
        - arrive_time: Array to store the arrival times of the customers.
        - num: The total number of customers to simulate.
        - interarrival: Dictionary holding the possible interarrival times and their probabilities.
        - counter: Dictionary holding the possible service times at the counter and their probabilities.
        - checkout: Dictionary holding the possible service times at checkout and their probabilities.
        - time_service_begin: Dictionary to store the start time of service for each customer at counter and checkout.
        - time_service_end: Dictionary to store the end time of service for each customer at counter and checkout.

        The constructor also initializes the arrival, counter, and checkout times by calling the respective methods.
        """
        self.checkout_time = None
        self.counter_time = None
        self.arrive_time = None

        self.num = num
        self.interarrival = {'time':np.arange(1,9), 'p':np.array([0.05,0.1,0.15,0.2,0.2,0.15,0.1,0.05])}
        self.counter = {'time':np.arange(1,7), 'p':np.array([0.1,0.15,0.2,0.3,0.2,0.05])}
        self.checkout = {'time':np.arange(1,5), 'p':np.array([0.2,0.4,0.3,0.1])}

        self.time_service_begin = {'counter':np.zeros(self.num), 'checkout':np.zeros(self.num)}
        self.time_service_end = {'counter':np.zeros(self.num), 'checkout':np.zeros(self.num)}

        self.get_arrive_time(num)
        self.get_counter_time(num)
        self.get_checkout_time(num)

        get_divider(1)

    def get_arrive_time(self, length):
        """
        Generates and sets the arrival times for all customers based on the specified interarrival times and probabilities.

        Parameters:
        - length: The number of customers to generate arrival times for.
        """
        t = np.zeros(length)
        for i in range(1, length):
            t[i] = np.random.choice(self.interarrival['time'], p=self.interarrival['p']) + t[i-1]
        self.arrive_time = t

    def get_counter_time(self, length):
        """
        Generates and sets the service times at the counter for all customers based on the specified service times and probabilities.

        Parameters:
        - length: The number of customers to generate service times for.
        """
        t = np.zeros(length)
        for i in range(length):
            t[i] = np.random.choice(self.counter['time'], p=self.counter['p'])
        self.counter_time = t

    def get_checkout_time(self, length):
        """
        Generates and sets the service times at checkout for all customers based on the specified service times and probabilities.

        Parameters:
        - length: The number of customers to generate service times for.
        """
        t = np.zeros(length)
        for i in range(length):
            t[i] = np.random.choice(self.checkout['time'], p=self.checkout['p'])
        self.checkout_time = t

    def get_wait(self):
        """
        Calculates the waiting times and idle times for customers at both the counter and checkout stages.

        This method iterates through each customer, determining the time they begin and end receiving service
        at both the counter and checkout. It calculates whether each customer has to wait at these stages,
        along with the actual waiting times. Additionally, it computes the idle times for both the counter
        and checkout, indicating periods when these service points are not serving any customers.

        The method also normalizes the idle times by the total time span from the first customer's arrival
        to the last customer's arrival, providing a relative measure of idle time in the context of the
        overall simulation period. These normalized idle times are then rounded to four decimal places.

        Returns:
            wait_time_counter (numpy.ndarray): Array of waiting times for each customer at the counter.
            wait_time_checkout (numpy.ndarray): Array of waiting times for each customer at checkout.
            wait_counter (numpy.ndarray): Binary array indicating if each customer had to wait at the counter (1) or not (0).
            wait_checkout (numpy.ndarray): Binary array indicating if each customer had to wait at checkout (1) or not (0).
            idle_time (dict): Dictionary with two keys ('counter' and 'checkout'), each containing the normalized and rounded
                              idle time as a proportion of the total simulation time for their respective service points.

        The method assumes that the first customer does not wait at the counter. It then proceeds to calculate
        for each following customer whether they need to wait based on the service end times of previous customers
        and their own arrival times. It similarly calculates for the checkout stage, taking into account the service
        end time at the counter for synchronization. Idle times are accumulated whenever a service point is waiting
        for the next customer, indicating efficiency gaps.
        """
        wait_counter = np.zeros(self.num)
        wait_checkout = np.zeros(self.num)
        wait_time_counter = np.zeros(self.num)
        wait_time_checkout = np.zeros(self.num)

        idle_time = {'counter':0,'checkout':0}

        self.time_service_end['counter'][0] = self.counter_time[0]

        self.time_service_begin['checkout'][0] = self.time_service_end['counter'][0]
        self.time_service_end['checkout'][0] = self.time_service_begin['checkout'][0] + self.checkout_time[0]
        for i in range(1, self.num):
            self.time_service_begin['counter'][i] = max(self.arrive_time[i], self.time_service_end['counter'][i-1])
            if self.time_service_end['counter'][i-1] > self.arrive_time[i]:
                wait_time_counter[i] = self.time_service_end['counter'][i-1] - self.arrive_time[i]
                wait_counter[i] = 1
            else:
                wait_time_counter[i] = 0
                wait_counter[i] = 0
                idle_time['counter'] += self.arrive_time[i] - self.time_service_end['counter'][i-1]

            self.time_service_end['counter'][i] = self.time_service_begin['counter'][i] + self.counter_time[i]

            self.time_service_begin['checkout'][i] = max(self.time_service_end['counter'][i],
                                                          self.time_service_end['checkout'][i-1])
            if self.time_service_end['checkout'][i-1] > self.time_service_end['counter'][i]:
                wait_time_checkout[i] = self.time_service_end['checkout'][i-1] - self.time_service_end['counter'][i]
                wait_checkout[i] = 1
            else:
                wait_time_checkout[i] = 0
                wait_checkout[i] = 0
                idle_time['checkout'] += self.time_service_end['counter'][i] - self.time_service_end['checkout'][i-1]

            self.time_service_end['checkout'][i] = self.time_service_begin['checkout'][i] + self.checkout_time[i]


        idle_time['counter'] /= self.arrive_time[-1]
        idle_time['checkout'] /= self.arrive_time[-1]

        idle_time['counter'] = round(idle_time['counter'],4)
        idle_time['checkout'] = round(idle_time['checkout'],4)


        return wait_time_counter, wait_time_checkout, wait_counter, wait_checkout, idle_time
```

In [ ]:

```
problem = Problem1(5000)

w_time_counter, w_time_checkout, w_counter, w_checkout, idle_time = problem.get_wait()

print("The average waiting time for counter of 5000 simulations is {}".format(np.mean(w_time_counter)))
print("The probability that a customer waits in the line for service at the counter of 5000 simulations is {}".format(np.mean(w_counter)))
print("The average waiting time for checkout of 5000 simulations is {}".format(np.mean(w_time_checkout)))
print("The probability that a customer waits in the line for service at the checkout of 5000 simulations is {}".format(np.mean(w_checkout)))
print("The probability that a customer waits at all of 5000 simulations is {}".format(np.mean(np.bitwise_or(w_checkout.astype(int), w_counter.astype(int)))))
print("The fraction of the time each server is idle of 5000 simulations is {}".format(idle_time))
```

```
**************************************************** Problem 1 ****************************************************
The average waiting time for counter of 5000 simulations is 1.4204
The probability that a customer waits in the line for service at the counter of 5000 simulations is 0.448
The average waiting time for checkout of 5000 simulations is 0.1956
The probability that a customer waits in the line for service at the checkout of 5000 simulations is 0.1268
The probability that a customer waits at all of 5000 simulations is 0.4946
The fraction of the time each server is idle of 5000 simulations is {'counter': 0.2181, 'checkout': 0.4897}
```

Result

(a) The average waiting time for counter of 5000 simulations is 1.4728

The probability that a customer waits in the line for service at the counter of 5000 simulations is 0.4694

(b) The average waiting time for checkout of 5000 simulations is 0.1792

The probability that a customer waits in the line for service at the checkout of 5000 simulations is 0.119

(c) The probability that a customer waits at all of 5000 simulations is 0.505

(d) The fraction of the time each server is idle of 5000 simulations is {'counter': 0.2089, 'checkout': 0.4882}

Problem #2

The `Problem2` class is designed to simulate a system where components have a certain lifetime and require replacement within a specified operation period. This class offers a detailed look into the dynamics of component replacement, including the calculation of lifetimes, replacement times, and the overall efficiency of the system.

Class Definition

`__init__(self, num)`

- Initializes an instance of `Problem2` with a specified number of simulations (`num`), allowing for multiple runs to gather statistically significant data.

`get_lifetime(self, mean=10)`

- Simulates component lifetimes based on an exponential distribution, with a default mean of 10 hours, converted to minutes for precision in calculations.

`get_replece_time(self, low=25, high=35)`

- Determines the time required to replace a component, drawing from a uniform distribution between specified low and high values, in minutes.

`simulate(self)`

- Orchestrates the simulation process, generating lifetimes and replacement times for components, and calculating the key metrics of the simulation, such as start and end times for replacements, and actual lifetimes considering a maximum cap.

Simulation Process

An instance of `Problem2` is created with 100 simulations to be performed. The simulation runs 10 iterations to collect data on replacement times and lifetimes. After each iteration, the results are aggregated into lists for analysis.

Analysis and Metrics

- **Average Total Time:** The average end time of the last replacement across all simulations, converted to hours, provides insight into the total operational duration.
- **Average Number of Replacements in First Five Days:** This metric calculates how many components need replacement within the first five days of operation, offering a measure of component durability and the frequency of maintenance required.
- **Average Component Usage Time:** The mean time a component is used before replacement gives an idea of component lifespan and efficiency.
- **Machine Idle Time Percentage:** This calculation shows the proportion of time the machine is not operational due to replacements, highlighting the efficiency and downtime of the system.

Conclusion

This detailed simulation and analysis through the `Problem2` class provide valuable insights into the operational efficiency, component durability, and maintenance needs of the system. By adjusting parameters and running multiple simulations, one can optimize the system for better performance and reduced downtime.

In [ ]:
```python
class Problem2:
    def __init__(self, num):
        """
        Initializes the Problem2 instance with a specified number of simulations.
        Parameters:
        - num: The total number of simulations to perform.
        """
        self.num = num
        get_divider(2)

    def get_lifetime(self, mean=10):
        """
        Generates and assigns lifetimes for each simulation based on an exponential distribution,
        multiplied by 60 to convert from hours to minutes.

        Parameters:
        - mean: The mean lifetime in hours before each simulation requires replacement. Default is 10 hours.
        """
        self.lifetime = -mean * np.log(np.random.uniform(size=self.num)) * 60

    def get_replece_time(self, low=25, high=35):
        """
        Generates and assigns replacement times for each simulation based on a uniform distribution.

        Parameters:
        - low: The minimum replacement time in minutes.
        - high: The maximum replacement time in minutes.
        """
        self.replece_time = np.random.uniform(low=low, high=high, size=self.num)

    def simulate(self):
        """
        Simulates the process by generating lifetimes and replacement times, then calculating the start
        and end times for replacements, along with the actual lifetimes considering a maximum lifetime limit of 15 hours (converted to minutes).

        Returns:
        - repalce_time_begin: List of start times for each replacement.
        - repalce_time_end: List of end times for each replacement.
        - actual_lifetime: List of actual lifetimes, limited to a maximum of 15 hours (900 minutes) for each simulation.
        """
        self.get_lifetime()
        self.get_replece_time()

        repalce_time_begin = [self.lifetime[0] if self.lifetime[0]<15*60 else 15*60]
        repalce_time_end = [repalce_time_begin[0] + self.replece_time[0]]
        actual_lifetime = [self.lifetime[0] if self.lifetime[0]<15*60 else 15*60]
        for i in range(1, self.num):
            repalce_time_begin.append(repalce_time_end[i-1]+(self.lifetime[i] if self.lifetime[i]<15*60 else 15*60))
            repalce_time_end.append(repalce_time_begin[i] + self.replece_time[i])
            actual_lifetime.append(self.lifetime[i] if self.lifetime[i]<15*60 else 15*60)
        return repalce_time_begin, repalce_time_end, actual_lifetime
```

In [ ]:

```
# Create an instance of Problem2 with 100 simulations
problem2 = Problem2(100)
# Initialize lists to store the results of multiple simulations
repalce_time_beginS = []
repalce_time_endS = []
actual_lifetimeS = []
# Run the simulate method 10 times to gather data
for i in range(10):
    repalce_time_begin, repalce_time_end, actual_lifetime = problem2.simulate()
    repalce_time_beginS.append(repalce_time_begin)
    repalce_time_endS.append(repalce_time_end)
    actual_lifetimeS.append(actual_lifetime)

print('The average total time of the simulations is {:.4f}h'.format(np.mean(np.array(repalce_time_endS)[:, -1])/60))
print('The average number of components that must be replaced during the first five days of operation is {}'.format(
    np.sum((np.array(repalce_time_beginS)<=5*24*60))/len(repalce_time_beginS)))
print("The average time (in hours) that a component is used is {:.4f}".format(np.mean(actual_lifetimeS)/60))
print("The average percent of the time the machine is idle is {:.4f}".format(1-np.sum(actual_lifetimeS)/ np.sum(np.array(repalce_time_endS)[:, -1])))
```

```
****************************************************** Problem 2 ******************************************************
The average total time of the simulations is 813.9486h
The average number of components that must be replaced during the first five days of operation is 15.0
The average time (in hours) that a component is used is 7.6387
The average percent of the time the machine is idle is 0.0615
```

Result

(a)The average total time of the simulations is 810.2510h

(b)The average number of components that must be replaced during the first five days of operation is 14.0

(c)The average time (in hours) that a component is used is 7.7732

(d)The average percent of the time the machine is idle is 0.0605

Problem #3

The `Problem3` class is a sophisticated simulation tool designed to model the operational and maintenance dynamics of a hypothetical system over a specified number of hours. It meticulously calculates the number of operational cycles and maintenance actions required, providing valuable insights into the system's reliability and efficiency.

Class Structure and Methods

Initialization: `__init__(self, hours)`

- **Purpose**: Initializes the simulation with a user-defined operational period.
- **Parameters**:
  - `hours`: Total simulation time in hours.
- **Functionality**: Sets up probability distributions for various events, such as replacement operations, running durations, intervals between calls, and failure occurrences.

Time Selection: `get_time(self, category)`

- **Purpose**: Randomly selects a time duration for a specified category based on predefined probabilities.
- **Parameters**:
  - `category`: Can be 'replace', 'run', or 'between_call', dictating the type of time interval to be randomly chosen.
- **Returns**: A time interval for the specified category.
- **Exception Handling**: Raises a ValueError if an unrecognized category is specified.

Failure Simulation: `fail(self)`

- **Purpose**: Simulates a failure event, determining whether a failure occurs based on a probabilistic model.
- **Returns**: A binary outcome where 1 indicates a failure and 0 indicates no failure.

Simulation Execution: `simulate(self)`

- **Purpose**: Conducts the simulation, tracking total operational time, the number of operational cycles, and the number of maintenance actions.
- **Operation**: Runs in a loop until the cumulative operational time exceeds the specified duration in minutes.
- **Returns**: The total number of operational cycles (`turn_on_times`) and maintenance actions (`replace_times`) during the simulation period.

Simulation Execution

An instance of `Problem3` is created with a specified operational duration of 50 hours, and the `simulate` method is invoked to model the system's behavior during this period. The simulation accounts for operational cycles, potential failures, and the necessary maintenance actions, providing a comprehensive overview of the system's performance and reliability.

Results Analysis

- **Operational Cycles**: The total number of times the system was activated or turned on during the simulation period.
- **Maintenance Actions**: The total number of times a maintenance action, such as a replacement, was required due to a failure.

The output from the simulation offers crucial insights into the operational efficiency and maintenance requirements of the system, allowing for better planning and optimization strategies to enhance performance and reliability.

In [ ]:
```
class Problem3:
    def __init__(self, hours):
        """
        Initializes the Problem3 instance with a specified number of hours for the simulation.

        Parameters:
        - hours: The total number of hours the simulation will run.

        The constructor also initializes several dictionaries to represent the probabilities of different time intervals:
        - replace: Time and probabilities for replacement operations.
        - run: Time and probabilities for running durations.
        - between_call: Time and probabilities for durations between calls.
        - failing: Values and probabilities to simulate failure occurrences.
        """
        self.hours = hours
        self.replace = {'time':np.arange(1,4), 'p':np.array([0.2,0.5,0.3])}
        self.run = {'time':np.arange(1,9), 'p':np.array([0.05,0.1,0.15,0.2,0.2,0.15,0.1,0.05])}
        self.between_call = {'time':np.arange(1,7), 'p':np.array([0.1,0.15,0.25,0.25,0.15,0.1])}
        self.failing = {'value':np.array([0,1]), 'p':np.array([0.6,0.4])}

        get_divider(3)

    def get_time(self, category):
        """
        Randomly selects a time based on the specified category and its associated probabilities.

        Parameters:
        - category: A string specifying the category of time to retrieve ('replace', 'run', or 'between_call').

        Returns:
        - A randomly chosen time from the specified category based on predefined probabilities.

        Raises:
        - ValueError: If an unknown category is passed.
        """
        if category == 'replace':
            return np.random.choice(self.replace['time'], p=self.replace['p'])
        elif category == 'run':
            return np.random.choice(self.run['time'], p=self.run['p'])
        elif category == 'between_call':
            return np.random.choice(self.between_call['time'], p=self.between_call['p'])
        else:
            raise ValueError('Unknown parameter {}'.format(category))

    def fail(self):
        """
        Simulates a failure event based on predefined probabilities.
        Returns:
        - A value indicating whether a failure occurred (1) or not (0).
        """
        return np.random.choice(self.failing['value'], p=self.failing['p'])

    def simulate(self):
        """
        Simulates the operation, tracking the total time, number of turn-on times, and number of replacements.

        The simulation runs until the total time exceeds the specified hours converted to minutes.

        Returns:
        - turn_on_times: The number of times the system was turned on during the simulation.
        - replace_times: The number of times a replacement was performed during the simulation.
        """
        total_time = 0
        turn_on_times = 0
        replace_times = 0
        while total_time <= self.hours*60:
            if self.fail() == 1:
                total_time += self.get_time('replace')
                replace_times += 1
            turn_on_times += 1
            total_time += self.get_time('run')
            total_time += self.get_time('between_call')
        return turn_on_times, replace_times
```

In [ ]:

```
problem3 = Problem3(50)
turn_on_times, replace_times = problem3.simulate()

print("The number of times the heater was called upon to turn on is {}".format(turn_on_times))
print("The number of times the coil on this heater had to be replaced is {}".format(replace_times))
```

```
****************************************************** Problem 3 ******************************************************
The number of times the heater was called upon to turn on is 334
The number of times the coil on this heater had to be replaced is 144
```

Result

(a) The number of times the heater was called upon to turn on is 343

(b) The number of times the coil on this heater had to be replaced is 140

Problem #4

The `Problem4` class is designed to simulate a service system, such as a queue in a bank or a restaurant, focusing on customer interarrival and service times during different parts of the day. It provides a detailed analysis of customer wait times, service efficiency, and system idle times.

Features and Methodology

- **Initialization (`__init__`)**: Sets up probability distributions for customer interarrival times for morning and afternoon, as well as service times, to closely mimic real-world scenarios.
- **Time Selection (`get_time`)**: Dynamically selects a random time for a given category (morning, afternoon, or service) based on the associated probability distributions. It ensures variability in the simulation, reflecting the randomness of real service systems.
- **Arrival Time Calculation (`get_arrival_time`)**: Generates a sequential list of customer arrival times throughout the operational day, transitioning from morning to afternoon schedules. This method models the fluctuating customer flow that many service systems experience.

- **Simulation Execution (`simulate`)**: Conducts the core simulation, tracking wait times, service initiation and completion times, and periods of inactivity. It offers a comprehensive view of the operational dynamics, including how long customers wait, how long they spend in the system, and the efficiency of the service provided.

Simulation Insights

Upon executing the `simulate` method, the class yields several key metrics:

- **Probability of Customer Wait**: Indicates how frequently customers need to wait in line before receiving service. This metric helps assess the demand and supply balance within the service system.
- **Average Wait Time**: Provides the mean duration that customers wait in line, offering insights into service accessibility and efficiency.
- **Average Wait Time for Waiting Customers**: Focuses on customers who actually waited, giving a deeper understanding of the wait experience for those not immediately served.
- **Average Time Spent in System**: Calculates the total time customers spend in the system, combining both wait and service times. This metric reflects the overall customer experience.
- **Server Idle Fraction**: Measures the proportion of operational time the server remains idle, indicating the system's utilization and efficiency.
- **Daily Customer Volume**: Counts the number of customers entering the system within a day, helping in capacity planning and staffing.

Conclusion

Through its simulation, `Problem4` offers valuable insights into the operational efficiency of service systems, highlighting areas for improvement in customer service and resource allocation. By analyzing wait times, service durations, and idle times, stakeholders can make informed decisions to enhance customer satisfaction and operational efficiency.

In [ ]:
```
class Problem4:
    def __init__(self):
        """
        Initializes the Problem4 instance with probability distributions for customer interarrival times
        in the morning and afternoon, and for service times. These distributions are used to simulate
        a service system like a queue in a bank or a restaurant.
        """
        self.morning_interarrival = {'time':np.arange(1,9),
                                     'p':np.array([0.05,0.1,0.15,0.2,0.2,0.15,0.1,0.05])}
        self.afternoon_interarrival = {'time':np.arange(1,9),
                                       'p':np.array([0.25,0.2,0.175,0.125,0.1,0.075,0.05,0.025])}
        self.service = {'time':np.arange(1,7),
                        'p':np.array([0.1,0.2,0.3,0.25,0.1,0.05])}


        get_divider(4)

    def get_time(self, category):
        """
        Selects a random time based on the specified category (morning, afternoon, service) and its associated
        probability distribution.

        Parameters:
        - category: A string specifying the category of time to retrieve.

        Returns:
        - A randomly chosen time from the specified category based on predefined probabilities.

        Raises:
        - ValueError: If an unknown category is passed.
        """
        if category == 'morning':
            return np.random.choice(self.morning_interarrival['time'], p=self.morning_interarrival['p'])
        elif category == 'afternoon':
            return np.random.choice(self.afternoon_interarrival['time'], p=self.afternoon_interarrival['p'])
        elif category == 'service':
            return np.random.choice(self.service['time'], p=self.service['p'])
        else:
            raise ValueError("Unknown parameter {}".format(category))


    def get_arrival_time(self):
        """
        Generates a list of customer arrival times throughout the day, transitioning from morning to
        afternoon interarrival times after 4 hours.

        Returns:
        - A list of cumulative arrival times up until 9 hours (540 minutes) minus the last entry if it exceeds 9 hours.
        """
        arrival_time = [self.get_time('morning')]
        while arrival_time[-1] <= 4*60:
            arrival_time.append(arrival_time[-1]+self.get_time('morning'))

        while arrival_time[-1] <= 9*60:
            arrival_time.append(arrival_time[-1]+self.get_time('afternoon'))

        return arrival_time[:-1]


    def simulate(self):
        """
        Simulates the service system, calculating wait times, service start and end times, and idle times
        for each customer based on the arrival and service time distributions.

        Returns:
        - wait_time: A list of wait times for each customer.
        - service_time_begin: A list of service start times for each customer.
        - service_time_end: A list of service end times for each customer.
        - idle_time: A list of idle times, indicating periods of no service activity.
        """
        wait_time = [0]
        arrival_time = self.get_arrival_time()
        idle_time = [arrival_time[0]]
        service_time_begin = [arrival_time[0]]
        service_time_end = [service_time_begin[0]+self.get_time('service')]
        for i in range(1, len(arrival_time)):
            wait_time.append(0 if service_time_end[i-1] <= arrival_time[i] else service_time_end[i-1] - arrival_time[i])
            service_time_begin.append(arrival_time[i] + wait_time[i])
            service_time_end.append(service_time_begin[i] + self.get_time('service'))
            if service_time_end[i-1] <= arrival_time[i]:
                idle_time.append(arrival_time[i]-service_time_end[i-1])
            else:
                idle_time.append(0)

        return wait_time, service_time_begin, service_time_end, idle_time
```

In [ ]:

```
problem4 = Problem4()

wait_time, service_time_begin, service_time_end, idle_time = problem4.simulate()

print('The probability that a customer waits in the line for service is {:.4f}'.format(np.count_nonzero(wait_time)/len(wait_time)))
print('The average waiting time per customer is {:.4f}'.format(np.mean(wait_time)))

print('The average waiting time per customer who has to wait is {:.4f}'.format(
    np.mean(np.array(wait_time)[np.nonzero(wait_time)])))

print("The average time a customer spends in the system is {:.4f}".format(
    np.mean(np.array(service_time_end) - np.array(service_time_begin) + np.array(wait_time))))
```

```
print("The fraction of the time the server is idle is {:.4f}".format(np.sum(idle_time)/(9*60)))

print('The average number of customers that enter the system in 1 day is {}'.format(len(wait_time)))
```

```
**************************************************** Problem 4 ****************************************************
The probability that a customer waits in the line for service is 0.7655
The average waiting time per customer is 13.3655
The average waiting time per customer who has to wait is 17.4595
The average time a customer spends in the system is 16.7655
The fraction of the time the server is idle is 0.1389
The average number of customers that enter the system in 1 day is 145
```

Result

(a) The probability that a customer waits in the line for service is 0.7655

The average waiting time per customer is 13.3655

(b) The average waiting time per customer who has to wait is 17.4595

(c) The average time a customer spends in the system is 16.7655

(d) The fraction of the time the server is idle is 0.1389

(e) The average number of customers that enter the system in 1 day is 145

Problem #5

First Inequality: $1 - 2x - x^2 \geq 0$

For the quadratic equation $-x^2 - 2x + 1 = 0$:

- Coefficients are: $a = -1$, $b = -2$, $c = 1$.
- Discriminant (Δ): $\Delta = b^2 - 4ac = (-2)^2 - 4(-1)(1) = 4 + 4 = 8$.
- Since $\Delta > 0$, there are two distinct real roots.

Using the quadratic formula $x = \frac{-b\pm\sqrt{\Delta}}{2a}$, we get:

- $x = \frac{-(-2)\pm\sqrt{8}}{2(-1)}$
- $x = \frac{2\pm2\sqrt{2}}{-2}$
- The roots are $x = -1 - \sqrt{2}$ and $x = -1 + \sqrt{2}$.

Since $a < 0$, the parabola opens downwards, and the inequality $-x^2 - 2x + 1 \geq 0$ is satisfied between the roots, so we have $-1 - \sqrt{2} \leq x \leq -1 + \sqrt{2}$.

Second Inequality: $1 + x - x^2 \geq 0$

For the quadratic equation $-x^2 + x + 1 = 0$:

- Coefficients are: $a = -1$, $b = 1$, $c = 1$.
- Discriminant (Δ): $\Delta = b^2 - 4ac = 1^2 - 4(-1)(1) = 1 + 4 = 5$.
- Since $\Delta > 0$, there are two distinct real roots.

Using the quadratic formula $x = \frac{-b\pm\sqrt{\Delta}}{2a}$, we get:

- $x = \frac{-1\pm\sqrt{5}}{-2}$
- The roots are $x = \frac{1\pm\sqrt{5}}{2}$.

Again, since $a < 0$, the parabola opens downwards, and the inequality $-x^2 + x + 1 \geq 0$ is satisfied between the roots, so we have $\frac{1-\sqrt{5}}{2} \leq x \leq \frac{1+\sqrt{5}}{2}$.

Intersection of Solutions

To find the solution to the system, we need the intersection of the two intervals:

- $-1-\sqrt{2} \leq x \leq -1+\sqrt{2}$
- $\frac{1-\sqrt{5}}{2} \leq x \leq \frac{1+\sqrt{5}}{2}$

The intersection of these two intervals gives us the domain of x that satisfies both inequalities. Graphical or numerical methods can be used to find this intersection precisely.

Thus, the domain of x that satisfies both inequalities is the intersection of the above two intervals.

In [ ]:
```
class Problem5:
    def __init__(self, num, low=-1-2**0.5, high=(1+5**0.5)/2):
        """
        Initializes the Problem5 instance with the number of simulations and the range for x values.
        The range is determined based on the intersection of solutions from two quadratic inequalities.

        Parameters:
        - num: The total number of simulations to run.
        - low: The lower bound for the x values, defaulting to the left endpoint of the interval
               satisfying the inequalities from a previous problem.
        - high: The upper bound for the x values, defaulting to the right endpoint of the interval
                satisfying the inequalities from a previous problem.
        """
        self.num = num
        self.low = low
        self.high = high

        get_divider(5)

    def get_x(self):
        """
        Generates a random x value uniformly distributed between the low and high attributes.

        Returns:
        - A random x value within the specified range.
        """
        return np.random.uniform(low=self.low, high=self.high)

    def get_y(self, x):
        """
        Generates a random y value based on the maximum of two functions of x, scaled by a random
        number between 0 and 1 to ensure it lies within the area under the curve defined by the maximum function.

        Parameters:
        - x: The x value for which to calculate y.

        Returns:
        - A random y value that is less than or equal to the maximum of the two functions of x.
        """
        return (max(1-2*x-x**2, 1+x-x**2)) * np.random.uniform()
    def get_z(self, x, y):
        """
        Generates a random y value based on the maximum of two functions of x, scaled by a random
        number between 0 and 1 to ensure it lies within the area under the curve defined by the maximum function.

        Parameters:
        - x: The x value for which to calculate y.

        Returns:
        - A random y value that is less than or equal to the maximum of the two functions of x.
        """
        return x**2 + (y-1)**3

    def simulate(self):
        """
        Runs the simulation to find the maximum z value over a series of randomly generated x and y pairs.
        It iterates through a number of simulations defined by num, and keeps track of the highest z value found.

        Returns:
        - The maximum z value found during the simulation.
        """
        max = -1000000
        for i in range(self.num):
            x = self.get_x()
            y = self.get_y(x)
            z = self.get_z(x, y)
            if z > max:
                max = z

        return max
```

In [ ]:

```
problem5 = Problem5(5000)

print("The max is {:.4f}".format(problem5.simulate()))
```

```
***************************************************** Problem 5 *****************************************************
The max is 4.9649
```

Result

The max is 4.9649

Problem #6

In [ ]:

```
import math
def get_x():
    """
    Generates a random x value uniformly distributed between 0 and 1.

    Returns:
    - A random x value.
    """
    return np.random.uniform()

def get_y(x):
    """
    Generates a random y value uniformly distributed between 0 and the square root of (1 - x^2)

    Parameters:
    - x: The x value used to calculate the upper bound for y.
    Returns:
    - A random y value within the specified range.
    """
    return np.random.uniform(low=0, high=(1-x**2)**0.5)

def get_z(x, y):
    """
    Generates a random z value uniformly distributed between 0 and the square root of (1 - x^2 - y^2),
    which is the upper bound of z given the constraint x^2 + y^2 + z^2 <= 1.

    Parameters:
    - x: The x value used in the constraint for z.
    - y: The y value used in the constraint for z.

    Returns:
    - A random z value within the specified range.
    """
    return np.random.uniform(low=0, high=1-x**2-y**2)

def get_value(x, y, z):
    """
    Calculates the value of the function (x + y + z) * exp(x * y * z).

    Parameters:
    - x: The x value used in the function.
    - y: The y value used in the function.
    - z: The z value used in the function.

    Returns:
    - The value calculated from x, y, and z.
    """
    return (x+y+z)*(math.exp(x*y*z))

class Problem6:
    def __init__(self, num):
        """
        Initializes the Problem6 instance with the number of simulations to run.

        Parameters:
        - num: The total number of simulations to run.
        """
        self.num = num

        get_divider(6)

    def simulate(self):
        """
        Runs the simulation to estimate the mean of the function (x + y + z) * exp(x * y * z)
        over a 3D volume within the unit sphere, accounting for the probabilistic weight
        of each random point based on its position.

        Returns:
        - The mean value of the function over the number of simulations.
        """
        values = []
        for i in range(self.num):
            x = get_x()
            y = get_y(x)
            z = get_z(x, y)
            values.append(get_value(x, y, z)*((1-x**2)**0.5)*(1-x**2-y**2))
        return np.mean(values)
```

In [ ]:
```
problem6 = Problem6(5000)

print("The result of 5000 Monte-Carlo Simulations is {:.4f}".format(problem6.simulate()))
```

```
***************************************************** Problem 6 *****************************************************
The result of 5000 Monte-Carlo Simulations is 0.4157
```

Result

The result of 5000 Monte-Carlo Simulations is 0.4157