МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

Кафедра дискретной математики и информационных технологий

ЛАБОРАТОРНАЯ РАБОТА №7

студента 4 курса 421 группы

направления 09.03.01 Информатика и вычислительная техника

факультета компьютерных наук и информационных технологий

Морозова Никиты Андреевича

Преподаватель

Станкевич Елена Петровна

Саратов 2024

Задача 10. Дана СМО типа $M|M|1$. Предполагается, что через экспоненциально распределенные интервалы времени в этой системе мгновенно уничтожаются все требования. После этого события система обслуживания продолжает нормальное функционирование. Построить имитационную модель системы. На основании 1000 выборочных значений оценить $\overline{u}$ и $\overline{n}$.

```
import heapq
import random
import numpy as np
import math

class Event:
    def __init__(self, time, event_type, data=None):
        self.time = time
        self.event_type = event_type  # Тип события:
                        # 'arrival', 'start_serving', 'departure', 'reset'
        self.data = data          # Дополнительные данные (departure_time)

    # Метод для сравнения событий по времени (необходим для работы с heapq)
    def __lt__(self, other):
        return self.time < other.time

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

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.items:
            return self.items.pop(0)
        return None

    def is_empty(self):
        return len(self.items) == 0

    def reset(self):
        self.items = []

class MM1SystemWithReset:
    def __init__(self, lambda_, mu, gamma):
        self.lambda_ = lambda_  # Интенсивность входящего потока требований
        self.mu = mu            # Интенсивность обслуживания
        self.gamma = gamma      # Интенсивность сбросов

        self.current_time = 0.0
        self.server_busy = False
        self.queue = Queue()      # Очередь требований
        self.event_queue = []     # Очередь событий (мини-куча)

        self.reset_times = []
        self.served_customers = []  # Список обслуженных требований
        self.reset_count = 0
```

```
    self.time_last_event = 0.0  # Время последнего события
    self.area_n = 0.0        # Интеграл числа требований по времени
    self.current_n = 0       # Текущее количество требований в системе


    first_arrival = self.current_time + generate_time(self.lambda_)
    heapq.heappush(self.event_queue, Event(first_arrival, 'arrival'))

    first_reset = self.current_time + generate_time(self.gamma)
    heapq.heappush(self.event_queue, Event(first_reset, 'reset'))

def handle_event(self, event):
    self.area_n += self.current_n * (event.time - self.time_last_event)
    self.time_last_event = event.time
    self.current_time = event.time

    if event.event_type == 'arrival':
        self.current_n += 1
        customer = {'arrival_time': event.time}
        self.queue.push(customer)

        # Планирование следующего прибытия
        next_arrival = self.current_time + generate_time(self.lambda_)
        heapq.heappush(self.event_queue, Event(next_arrival, 'arrival'))

        if not self.server_busy:
            self.start_serving()

    elif event.event_type == 'start_serving':
        self.server_busy = True
        customer = self.queue.pop()
        customer['service_start'] = self.current_time
        # Генерация времени окончания обслуживания
        service_time = generate_time(self.mu)
        heapq.heappush(self.event_queue,
                       Event(self.current_time + service_time, 'departure', customer))

    elif event.event_type == 'departure':
        self.server_busy = False
        self.current_n -= 1
        customer = event.data
        customer['departure_time'] = self.current_time
        self.served_customers.append(customer)

        if not self.queue.is_empty():
            self.start_serving()

    elif event.event_type == 'reset':
        self.reset_count += 1
        self.queue.reset()
        if self.server_busy:
            self.server_busy = False
        self.current_n = 0

        # Планирование следующего сброса
        next_reset = self.current_time + generate_time(self.gamma)
        heapq.heappush(self.event_queue, Event(next_reset, 'reset'))
```

```
    def start_serving(self):
        if not self.queue.is_empty() and not self.server_busy:
            heapq.heappush(self.event_queue,
                           Event(self.current_time, 'start_serving'))

    def run(self, num_customers=1000):
        while len(self.served_customers) < num_customers:
            if not self.event_queue:
                break  # Если событий нет
            event = heapq.heappop(self.event_queue)
            self.handle_event(event)

        # Среднее время пребывания (u)
        u_times = [c['departure_time'] - c['arrival_time'] for c in self.served_customers]
        u = np.sum(u_times) / len(u_times)

        # Среднее число требований (n)
        total_time = self.current_time
        n = self.area_n / total_time if total_time > 0 else 0

        return u, n


def generate_time(lambda_):
    return -(1/lambda_) * math.log(random.uniform(0, 1))


# Пример использования
if __name__ == "__main__":
    lambda_ = 0.5  # Интенсивность поступления требований
    mu = 1.0       # Интенсивность обслуживания
    gamma = 0.1    # Интенсивность сбросов

    if lambda_ >= mu:
        print("Ошибка: система нестабильна (lambda >= mu)")
    else:
        system = MM1SystemWithReset(lambda_, mu, gamma)
        u, n = system.run(num_customers=1000)

        print(f"Среднее время пребывания (u): {u:.4f}")
        print(f"Среднее число требований (n): {n:.4f}")
```

```
Среднее время пребывания (u): 1.5609
Среднее число требований (n): 0.2856
```

Представим теперь метод вычисления $\tilde{n}$ – оценки математического ожидания числа требований в системе обслуживания. Пусть T – длительность модельного времени проведения эксперимента с моделью, $\tau_i^{(k)}$ – длительность i-го интервала времени, когда в системе обслуживания находилось ровно k требований, $k = 0, 1, 2, \ldots$, $i = 1, 2, \ldots, L_k$, где R_k – число интервалов времени, когда в СМО находилось ровно k требований.

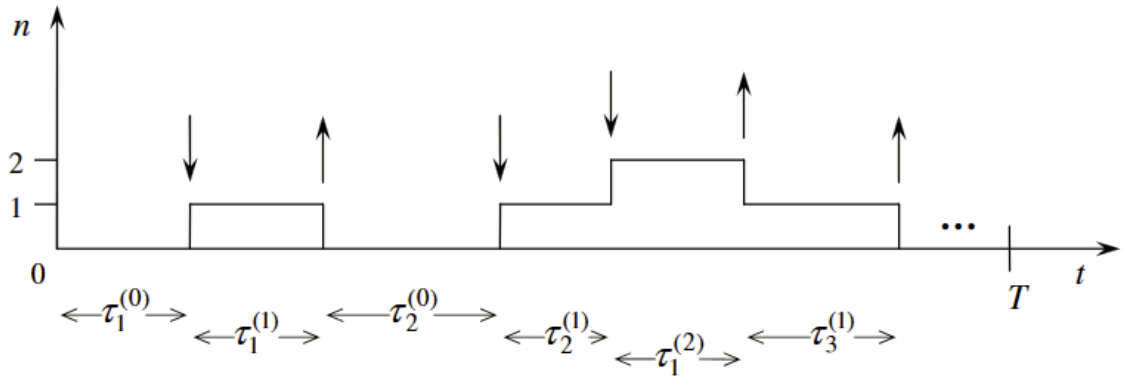

Рис. 3.3. Реализация процесса поступления и обслуживания требований

На рис. 3.3 показана одна из типичных реализаций процесса поступления и обслуживания требований системой обслуживания. На оси модельного времени t вертикальными стрелками, направленными вниз и вверх, показаны моменты соответственно поступления и ухода требований из СМО. Буквой n обозначено число требований в СМО, $\tau_1^{(0)}$, $\tau_2^{(0)}$, … – интервалы времени, в течение которых в СМО нет требований, $\tau_1^{(1)}$, $\tau_2^{(1)}$, $\tau_3^{(1)}$, … – интервалы времени, в течение которых в СМО находится одно требование, $\tau_1^{(2)}$, … – интервалы времени, в течение которых в СМО находится два требования.

Оценка вероятности того, что в системе обслуживания находится ровно k требований, равна

$$\tilde{p}_k = \frac{1}{T}\sum_{i=1}^{R_k} \tau_i^{(k)}, \quad k = 0, 1, 2, \ldots.$$

Тогда оценка величины $\bar{n}$ вычисляется по формуле

$$\tilde{n} = \sum_{k=1}^{\infty} k\tilde{p}_k .$$