src

problem_1.py

```
import math
import random

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches


print('Exercise 1')
print('')


################################################################################
#                                 Exercise 1) a)                               #
################################################################################


print('a) Run Kolmogorov-Smirnov test to determine distribution of time between client arriv


with open('../data/datosTP2EJ1.txt') as my_file:
    time_between_arrivals = sorted([float(line) for line in my_file.readlines()])


def exponential_cdf(x, scale):
    return 1 - math.e ** (- x / scale)


def kolmogorov_smirnov_test(scale):
    significance_level = 0.01
    max_distance = math.sqrt(
        (-1 / (2 * len(time_between_arrivals))) *
        math.log(significance_level / 2)
    )
    rejected = False
    for index, time in enumerate(time_between_arrivals):
        actual_value = exponential_cdf(time, scale=scale)
        lower_value = index / len(time_between_arrivals)
        upper_value = (index + 1) / len(time_between_arrivals)
        distance = max(
            abs(upper_value - actual_value),
            abs(lower_value - actual_value)
        )
        if distance > max_distance:
            rejected = True
            break
    if rejected:
        print(f'    Exponential distribution with scale of {scale} seconds discarded')
    else:
        print(f'    Exponential distribution with scale of {scale} seconds accepted')


kolmogorov_smirnov_test(scale=180)
kolmogorov_smirnov_test(scale=240)


################################################################################
#                                Exercise 1) b)                                #
################################################################################


print('b) Simulate 1000 days of new ATM')


seconds_in_hour = 3600
day_in_seconds = 24 * seconds_in_hour
thousand_days_in_seconds = 1000 * day_in_seconds

accepted_scale = 180
arrival_timestamps = []
last_arrival = 0
while last_arrival < thousand_days_in_seconds:
    last_arrival += np.random.exponential(scale=accepted_scale)
    arrival_timestamps.append(last_arrival)

max_banknote_count = 2000
balance = 0
next_available_timestamp = 0

successful_operations = []
unsuccessful_operations = []
is_first_day = True
next_day_timestamp = 0

total_client_time = 0
client_count = 0
success_count = 0
```

```
for arrival in arrival_timestamps:
    if next_available_timestamp > thousand_days_in_seconds:
        break
    if next_available_timestamp > day_in_seconds:
        is_first_day = False
    if arrival > next_day_timestamp:
        balance = max_banknote_count
        next_day_timestamp += day_in_seconds

    if random.random() < 0.25:
        duration = np.random.exponential(scale=300)
        amount = 10 + random.random() * 100
    else:
        duration = np.random.exponential(scale=90)
        amount = -(3 + random.random() * 47)

    wait_time = max(0, next_available_timestamp - arrival)
    next_available_timestamp = arrival + wait_time + duration
    total_client_time += wait_time + duration
    client_count += 1

    new_balance = balance + amount
    if 0 <= new_balance <= max_banknote_count:
        balance = new_balance
        success_count += 1
        if is_first_day:
            successful_operations.append((next_available_timestamp, balance))
    else:
        if is_first_day:
            unsuccessful_operations.append((next_available_timestamp, balance))


################################################################################
#                                Exercise 1) c)                                #
################################################################################


print('c) Graph banknotes in ATM in the first day')


marker_size = 1.1
plt.scatter([x[0] / seconds_in_hour for x in successful_operations],
            [y[1] for y in successful_operations], c='blue', s=marker_size)
plt.scatter([x[0] / seconds_in_hour for x in unsuccessful_operations],
            [y[1] for y in unsuccessful_operations], c='red', s=marker_size)
```

```
plt.title('Banknotes in ATM in the first day')
plt.xlabel('Time (hours)')
plt.xlim(0, 24)
plt.ylabel('Banknotes in ATM')
plt.ylim(0, max_banknote_count)
blue_patch = mpatches.Patch(color='blue', label='Successful operation')
red_patch = mpatches.Patch(color='red', label='Unsuccessful operation')
plt.legend(handles=[blue_patch, red_patch])


################################################################################
#                                Exercise 1) d)                                #
################################################################################


print(f'd) Mean wait + usage time: {total_client_time / client_count / 60:.1f} minutes')


################################################################################
#                                Exercise 1) e)                                #
################################################################################

current_failure_percentage = 20


print(f'e) Compare to current ATM (current failure rate: {current_failure_percentage}%)')


new_failure_percentage = (1 - success_count / client_count) * 100
if new_failure_percentage < current_failure_percentage:
    conclusion = 'ATM should be changed'
else:
    conclusion = 'ATM should not be changed'
print(f'    {conclusion} since new failure rate is {new_failure_percentage:.1f}%')


plt.show()
```

problem_2.py

```
import random
import numpy as np
import matplotlib.pyplot as plt
```

```
class Probabilities:
    def __init__(self, request_number):
        self.p = 1 / 40
        self.q = 1 / 30
        self.request_number = request_number
        self.probs = [
            {
                'prob': self.move_forward_prob(),
                'name': 'move_forward_prob'
            },
            {
                'prob': self.go_back_prob(),
                'name': 'go_back_prob',
            },
            {
                'prob': self.staying_prob(),
                'name': 'staying_prob'
            }
        ]
        self.probs = sorted(self.probs, key=lambda x: x['prob'])
        for idx in range(len(self.probs) - 1):
            self.probs[idx + 1]['prob'] += self.probs[idx]['prob']
        self.change_state = {
            'move_forward_prob': self.add_request,
            'go_back_prob': self.finish_request,
            'staying_prob': lambda: None
        }

    def next(self, action):
        self.change_state[action]()
        if self.request_number == 0:
            return ZeroRequestProbabilities(self.request_number)
        return GreaterZeroRequestProbabilities(self.request_number)

    def add_request(self):
        self.request_number += 1

    def finish_request(self):
        if self.request_number == 0:
            return
        self.request_number -= 1

    def staying_prob(self):
        raise Exception('not implemented')
    def move_forward_prob(self):
        raise Exception('not implemented')

    def go_back_prob(self):
        raise Exception('not implemented')


class ZeroRequestProbabilities(Probabilities):
    def staying_prob(self):
        return 1 - self.p

    def move_forward_prob(self):
        return self.p

    def go_back_prob(self):
        return 0


class GreaterZeroRequestProbabilities(Probabilities):
    def staying_prob(self):
        p = self.p
        q = self.q
        return p * q + (1 - p) * (1 - q)

    def move_forward_prob(self):
        return self.p * (1 - self.q)

    def go_back_prob(self):
        return self.q * (1 - self.p)


class Server:
    def __init__(self, total_time=1000000, time_between_arrivals=10):
        """
        total_time: total time in milli-seconds
        time_between_arrivals: time for arrival or processing completion
        in milli-seconds
        """
        self.total_time = total_time
        self.time_between_arrivals = time_between_arrivals
        self.occurrences_per_time = {}
        self.occurrences_per_state = {}
        self.no_processing_request_number = 0
        self.probabilities = ZeroRequestProbabilities(0)
    def graph_occurrences_per_state(self):
        x = list(self.occurrences_per_state.keys())
        y = list(self.occurrences_per_state.values())
        y_pos = np.arange(len(x))
        x_ticks = (str(t) for t in x)
        plt.bar(y_pos, y, color='C1', align='center', alpha=0.5, log=True)
        plt.xticks(y_pos, x_ticks)
        plt.title('occurrences_per_state')
        plt.show()

    def graph_occurrences_per_time(self):
        x = list(self.occurrences_per_time.keys())
        y = list(self.occurrences_per_time.values())
        plt.plot(x, y, color='C2', label='occurrences_per_time')
        plt.legend(loc='upper right')
        plt.show()

    def percentage_no_processing_request(self):
        return self.no_processing_request_number * \
                self.time_between_arrivals/self.total_time*100

    def log_stats(self, timestamp):
        request_number = self.probabilities.request_number
        self.occurrences_per_time.setdefault(timestamp, 0)
        self.occurrences_per_time[timestamp] += request_number
        self.occurrences_per_state.setdefault(request_number, 0)
        self.occurrences_per_state[request_number] += 1

    def transition(self, timestamp):
        arrival = random.uniform(0.00000, 1.00000)
        for idx, prob in enumerate(self.probabilities.probs):
            if arrival <= prob['prob']:
                if prob['name'] == 'staying_prob':
                    self.no_processing_request_number += 1
                self.probabilities = self.probabilities.next(prob['name'])
                self.log_stats(timestamp)
                return True
        return None


server = Server()
for i in range(100000):
    server.transition((i + 1) * 10)
server.graph_occurrences_per_time()
server.graph_occurrences_per_state()
print(f"percentage_no_processing_request: "
      f"{server.percentage_no_processing_request()}")
```

problem_3.py

problem_4.py