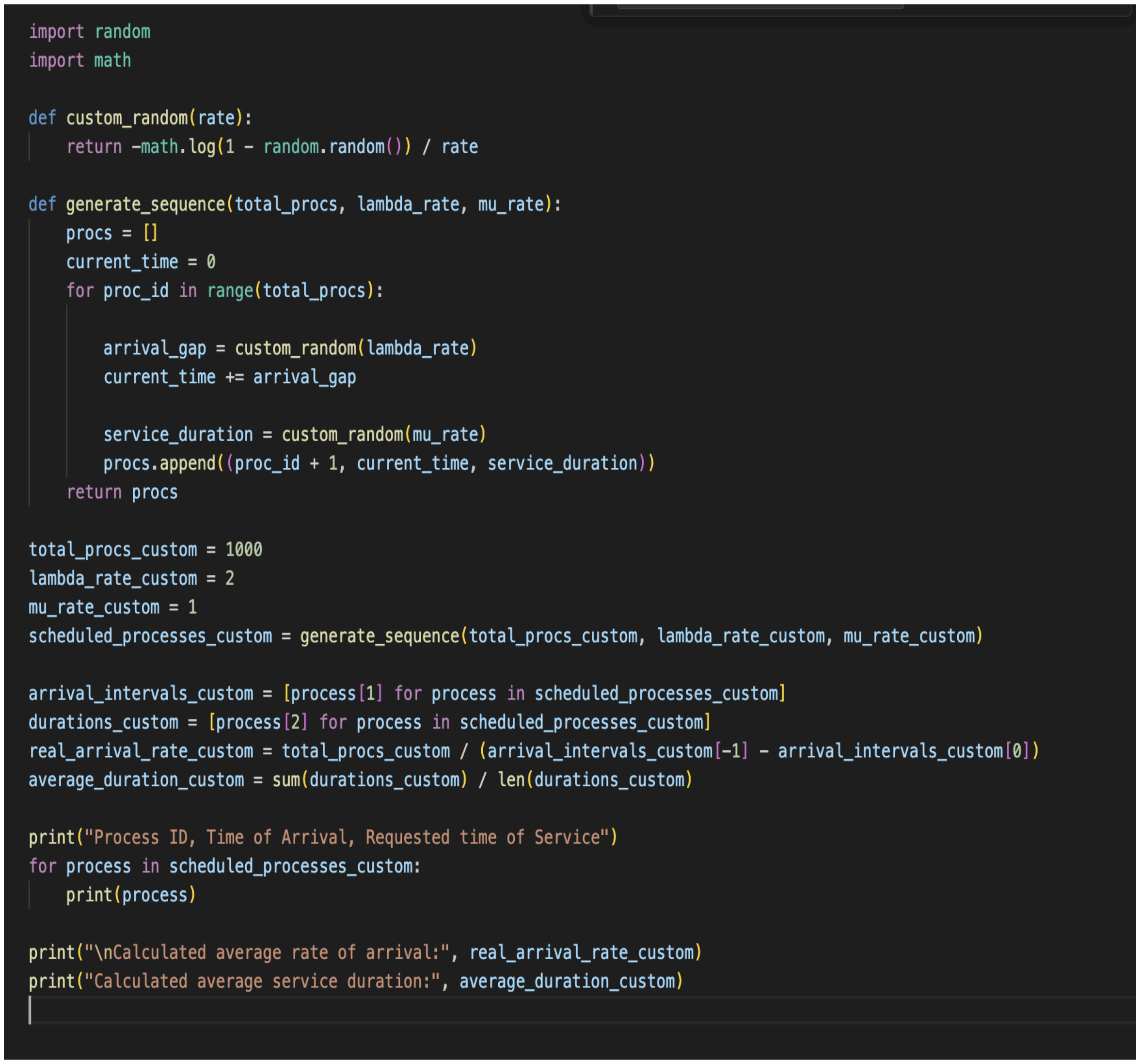
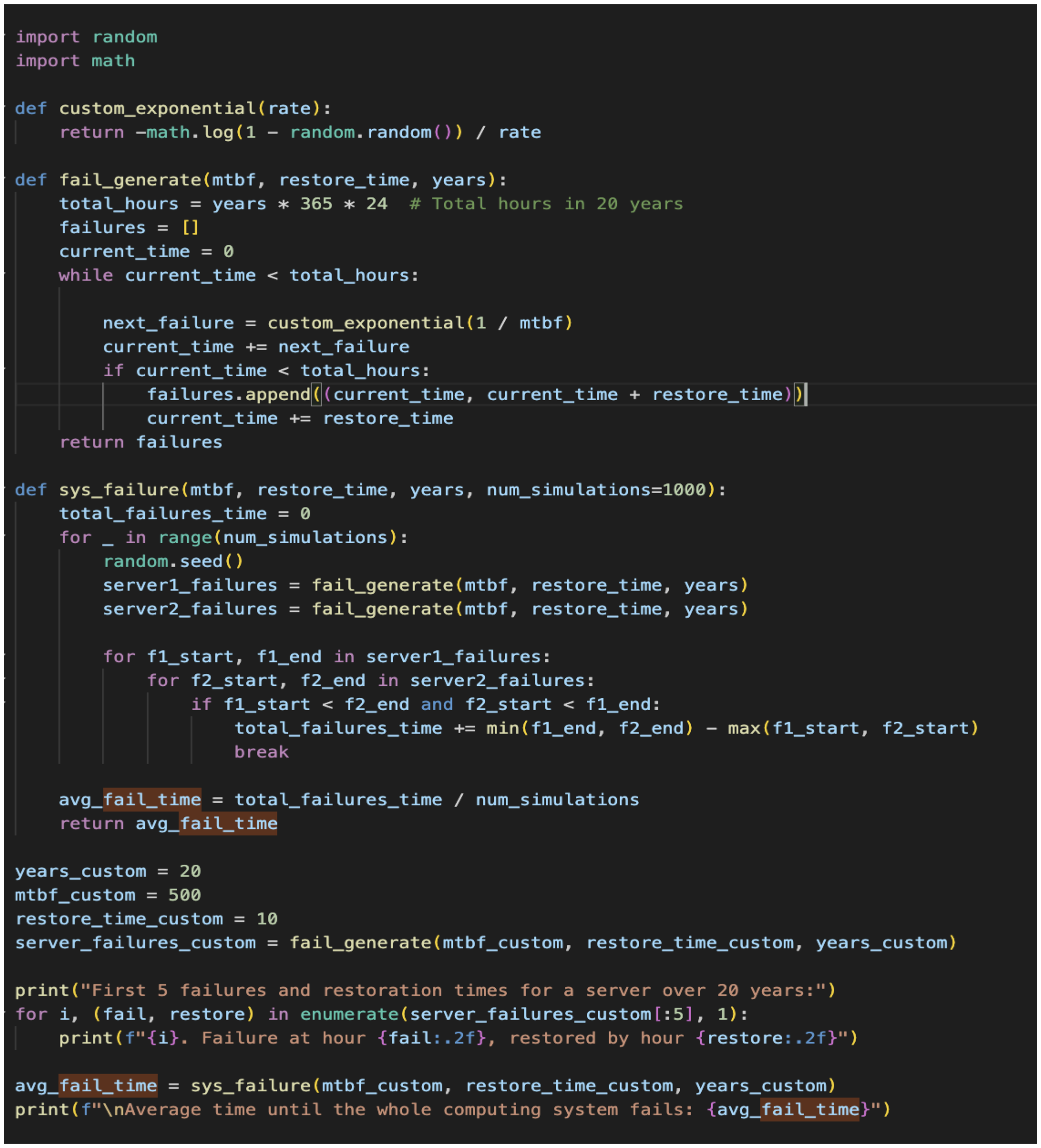
The purpose of this prompt is to guide the creation of a simulation that models the workload of a computer system, specifically focusing on process arrivals and their service times. This simulation is aimed at understanding and analyzing system behavior under a specified load, which is common in studies of operating systems, performance analysis, and capacity planning



```python
import random
import math

def custom_random(rate):
    return -math.log(1 - random.random()) / rate

def generate_sequence(total_procs, lambda_rate, mu_rate):
    procs = []
    current_time = 0
    for proc_id in range(total_procs):

        arrival_gap = custom_random(lambda_rate)
        current_time += arrival_gap

        service_duration = custom_random(mu_rate)
        procs.append((proc_id + 1, current_time, service_duration))
    return procs

total_procs_custom = 1000
lambda_rate_custom = 2
mu_rate_custom = 1
scheduled_processes_custom = generate_sequence(total_procs_custom, lambda_rate_custom, mu_rate_custom)

arrival_intervals_custom = [process[1] for process in scheduled_processes_custom]
durations_custom = [process[2] for process in scheduled_processes_custom]
real_arrival_rate_custom = total_procs_custom / (arrival_intervals_custom[-1] - arrival_intervals_custom[0])
average_duration_custom = sum(durations_custom) / len(durations_custom)

print("Process ID, Time of Arrival, Requested time of Service")
for process in scheduled_processes_custom:
    print(process)

print("\nCalculated average rate of arrival:", real_arrival_rate_custom)
print("Calculated average service duration:", average_duration_custom)
```

The purpose of this prompt is to design a simulation that models the reliability and failure dynamics of a redundant computing system. This simulation aims to provide insights into the effectiveness of redundancy as a fault tolerance strategy, specifically in scenarios where system components have defined probability of failure and recovery time.



```python
import random
import math

def custom_exponential(rate):
    return -math.log(1 - random.random()) / rate

def fail_generate(mtbf, restore_time, years):
    total_hours = years * 365 * 24  # Total hours in 20 years
    failures = []
    current_time = 0
    while current_time < total_hours:

        next_failure = custom_exponential(1 / mtbf)
        current_time += next_failure
        if current_time < total_hours:
            failures.append((current_time, current_time + restore_time))
            current_time += restore_time
    return failures

def sys_failure(mtbf, restore_time, years, num_simulations=1000):
    total_failures_time = 0
    for _ in range(num_simulations):
        random.seed()
        server1_failures = fail_generate(mtbf, restore_time, years)
        server2_failures = fail_generate(mtbf, restore_time, years)

        for f1_start, f1_end in server1_failures:
            for f2_start, f2_end in server2_failures:
                if f1_start < f2_end and f2_start < f1_end:
                    total_failures_time += min(f1_end, f2_end) - max(f1_start, f2_start)
                    break

    avg_fail_time = total_failures_time / num_simulations
    return avg_fail_time

years_custom = 20
mtbf_custom = 500
restore_time_custom = 10
server_failures_custom = fail_generate(mtbf_custom, restore_time_custom, years_custom)

print("First 5 failures and restoration times for a server over 20 years:")
for i, (fail, restore) in enumerate(server_failures_custom[:5], 1):
    print(f"{i}. Failure at hour {fail:.2f}, restored by hour {restore:.2f}")

avg_fail_time = sys_failure(mtbf_custom, restore_time_custom, years_custom)
print(f"\nAverage time until the whole computing system fails: {avg_fail_time}")
```