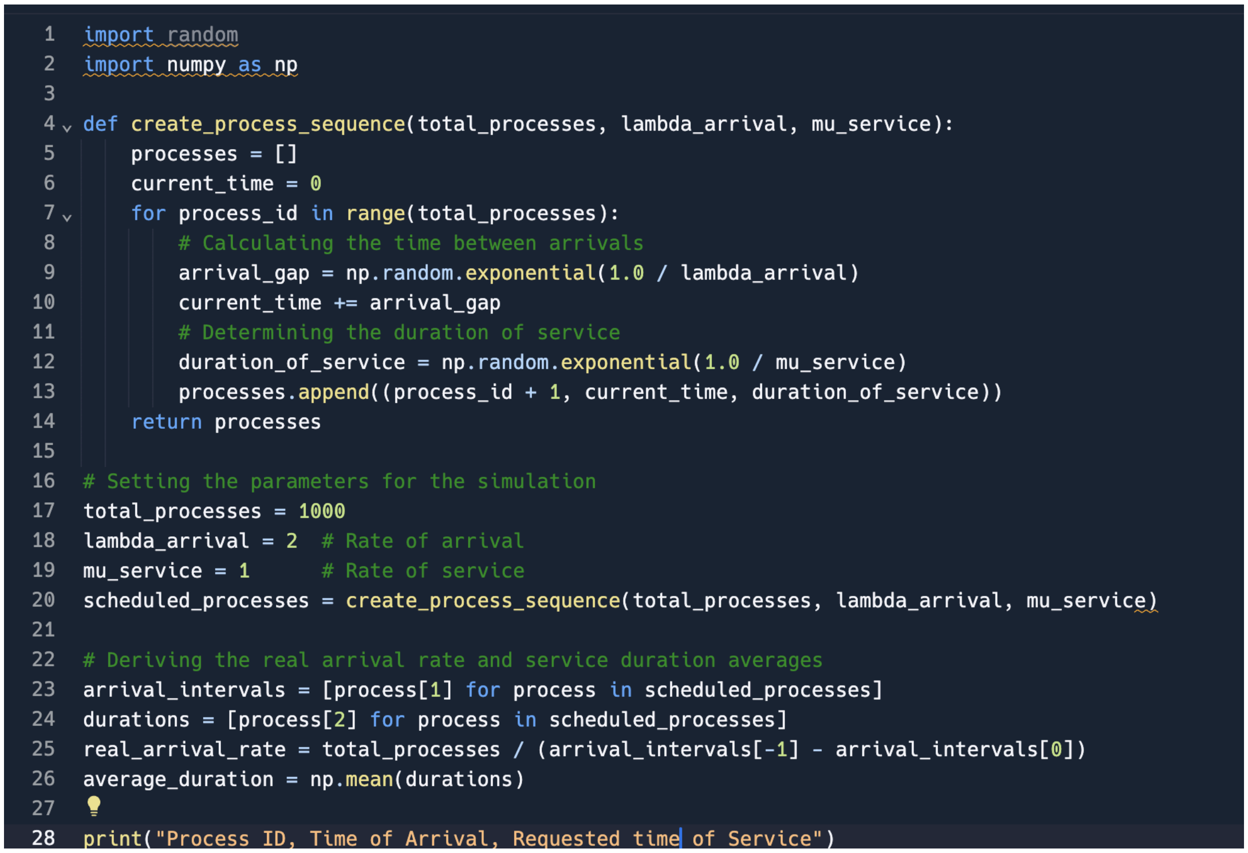
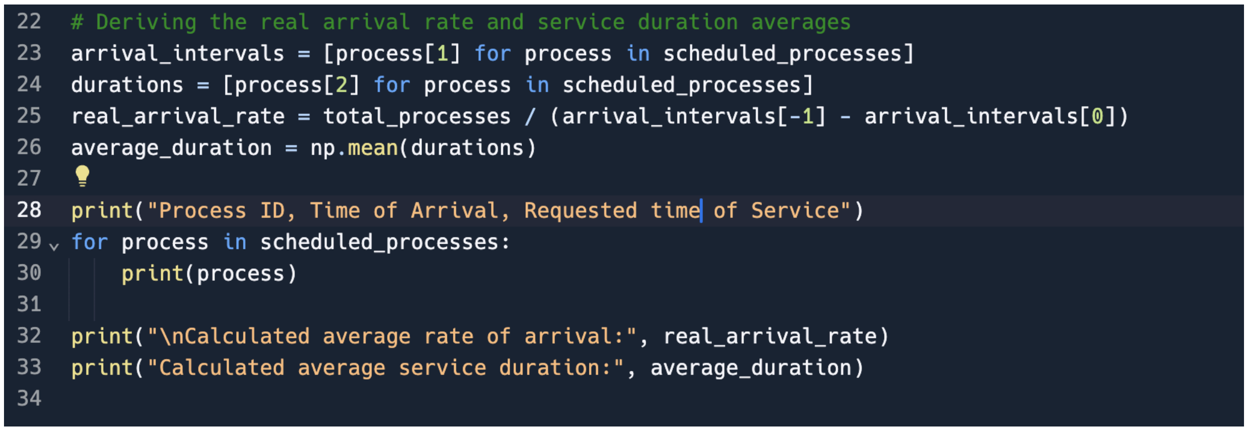
Problem 1

The program we're talking about acts like a virtual test for a computer system, creating a realistic *scenario where tasks arrive randomly and need different amounts of time to complete. Imagine it as simulating a busy day at an office, with workers (processes) coming in at unpredictable times and each needing a unique amount of attention (service time) to finish their job. After simulating a whole day, the program looks back to figure out, on average, how often someone walked in and how long it took to help each person. This helps understand how well the office (or in our case, the computer system) can keep up with the work, highlighting its efficiency and capability to manage the day's demands.*



```python
import random
import numpy as np

def create_process_sequence(total_processes, lambda_arrival, mu_service):
    processes = []
    current_time = 0
    for process_id in range(total_processes):
        # Calculating the time between arrivals
        arrival_gap = np.random.exponential(1.0 / lambda_arrival)
        current_time += arrival_gap
        # Determining the duration of service
        duration_of_service = np.random.exponential(1.0 / mu_service)
        processes.append((process_id + 1, current_time, duration_of_service))
    return processes

# Setting the parameters for the simulation
total_processes = 1000
lambda_arrival = 2  # Rate of arrival
mu_service = 1      # Rate of service
scheduled_processes = create_process_sequence(total_processes, lambda_arrival, mu_service)

# Deriving the real arrival rate and service duration averages
arrival_intervals = [process[1] for process in scheduled_processes]
durations = [process[2] for process in scheduled_processes]
real_arrival_rate = total_processes / (arrival_intervals[-1] - arrival_intervals[0])
average_duration = np.mean(durations)

print("Process ID, Time of Arrival, Requested time of Service")
```



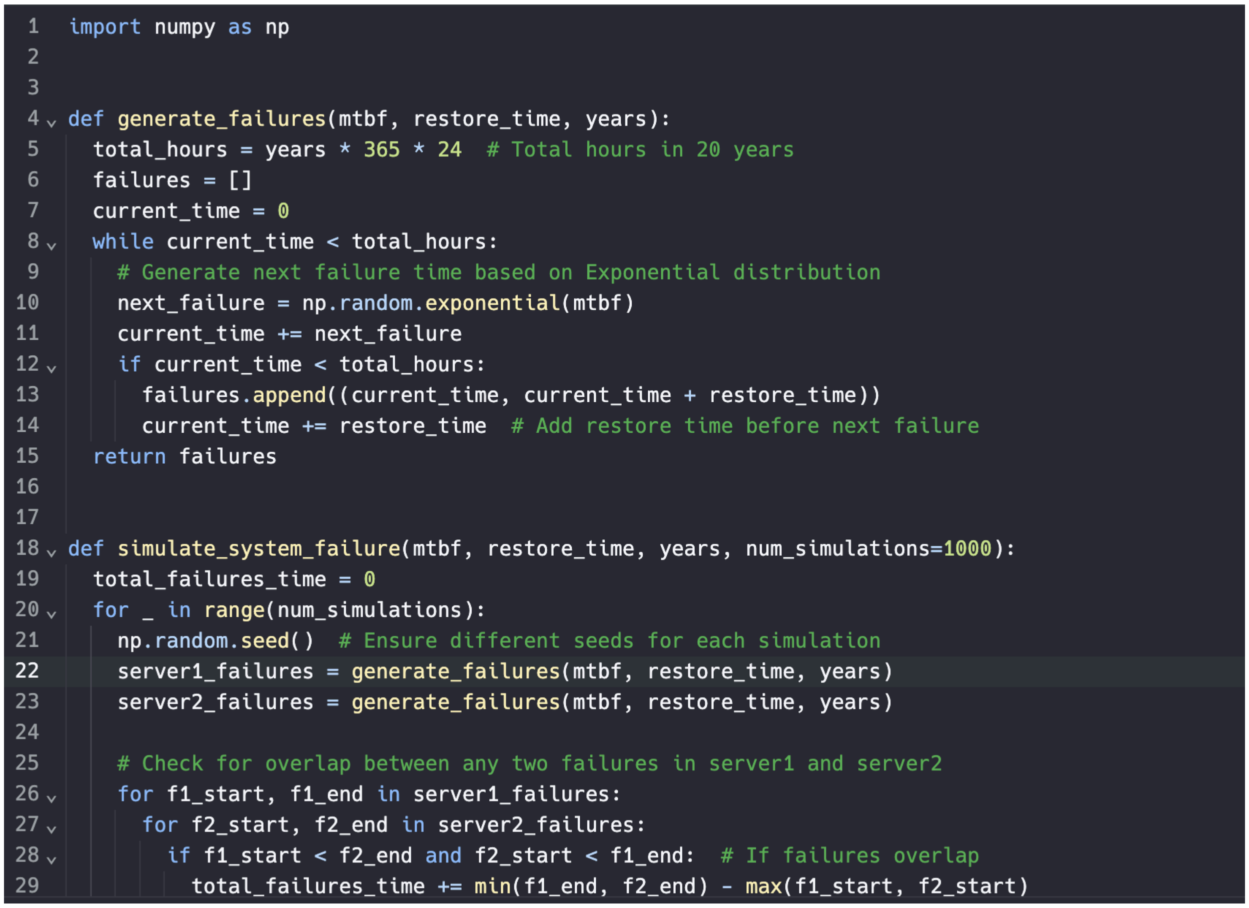
```python
# Deriving the real arrival rate and service duration averages
arrival_intervals = [process[1] for process in scheduled_processes]
durations = [process[2] for process in scheduled_processes]
real_arrival_rate = total_processes / (arrival_intervals[-1] - arrival_intervals[0])
average_duration = np.mean(durations)

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

print("\nCalculated average rate of arrival:", real_arrival_rate)
print("Calculated average service duration:", average_duration)

```

Problem 2

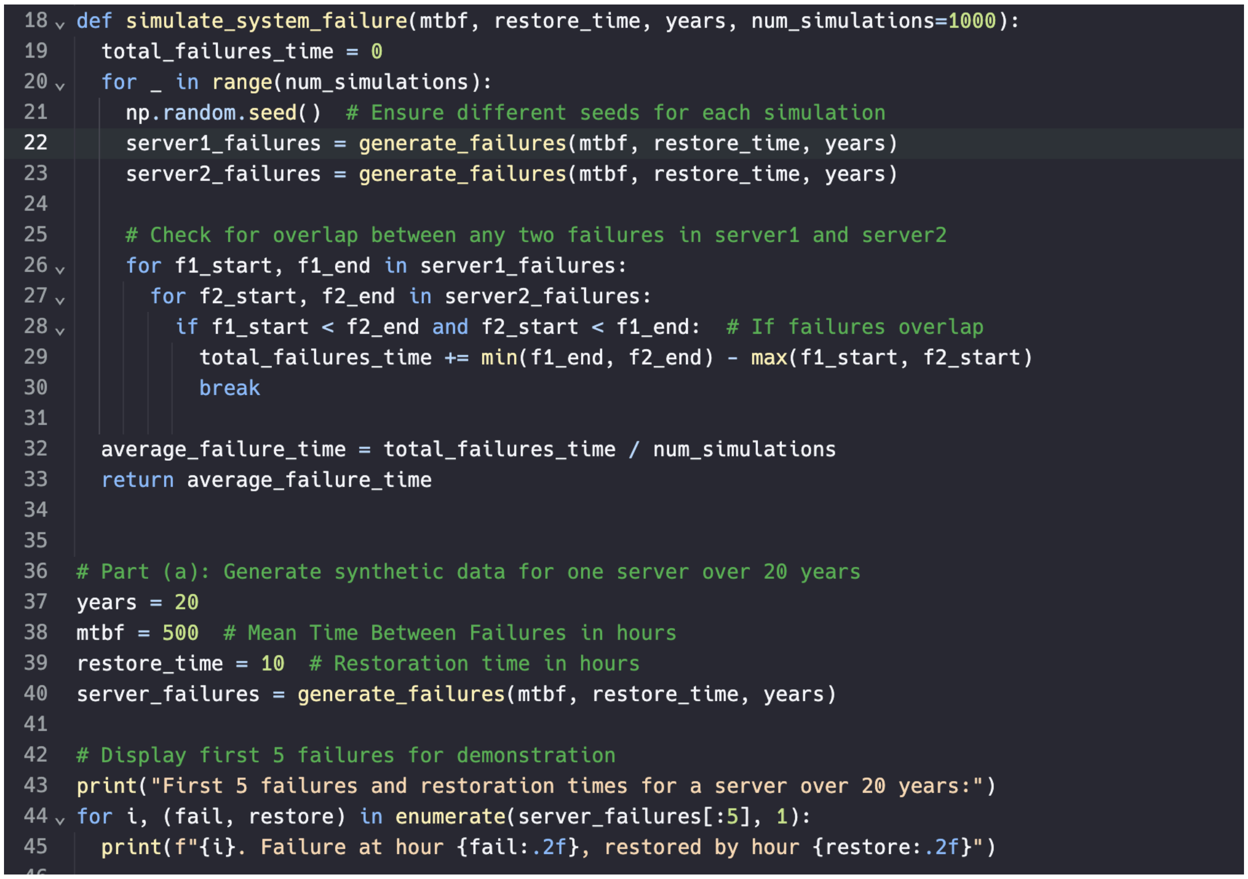
In Part A, our goal is to model the downtime and recovery periods of a server across a span of 20 years.

We presume that each server experiences an average time of 500 hours between failures, adhering to an Exponential Distribution for its operational intervals.

Following a failure, a restoration process is initiated, taking a fixed duration of 10 hours to synchronize data from its backup. This simulation is designed to evaluate the server's dependability and plan for its upkeep.



```python
import numpy as np


def generate_failures(mtbf, restore_time, years):
    total_hours = years * 365 * 24  # Total hours in 20 years
    failures = []
    current_time = 0
    while current_time < total_hours:
        # Generate next failure time based on Exponential distribution
        next_failure = np.random.exponential(mtbf)
        current_time += next_failure
        if current_time < total_hours:
            failures.append((current_time, current_time + restore_time))
            current_time += restore_time  # Add restore time before next failure
    return failures


def simulate_system_failure(mtbf, restore_time, years, num_simulations=1000):
    total_failures_time = 0
    for _ in range(num_simulations):
        np.random.seed()  # Ensure different seeds for each simulation
        server1_failures = generate_failures(mtbf, restore_time, years)
        server2_failures = generate_failures(mtbf, restore_time, years)

        # Check for overlap between any two failures in server1 and server2
        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:  # If failures overlap
                    total_failures_time += min(f1_end, f2_end) - max(f1_start, f2_start)
```



```python
def simulate_system_failure(mtbf, restore_time, years, num_simulations=1000):
  total_failures_time = 0
  for _ in range(num_simulations):
    np.random.seed()  # Ensure different seeds for each simulation
    server1_failures = generate_failures(mtbf, restore_time, years)
    server2_failures = generate_failures(mtbf, restore_time, years)

    # Check for overlap between any two failures in server1 and server2
    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:  # If failures overlap
          total_failures_time += min(f1_end, f2_end) - max(f1_start, f2_start)
          break

  average_failure_time = total_failures_time / num_simulations
  return average_failure_time


# Part (a): Generate synthetic data for one server over 20 years
years = 20
mtbf = 500  # Mean Time Between Failures in hours
restore_time = 10  # Restoration time in hours
server_failures = generate_failures(mtbf, restore_time, years)

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

The `generate_failures` function models the times at which a server fails, utilizing the Exponential Distribution with the specified Mean Time Between Failures (MTBF).

This function logs the moment of each failure and the time when the system is fully restored (10 hours subsequently), continuing until the entire simulation duration (20 years) is accounted for.

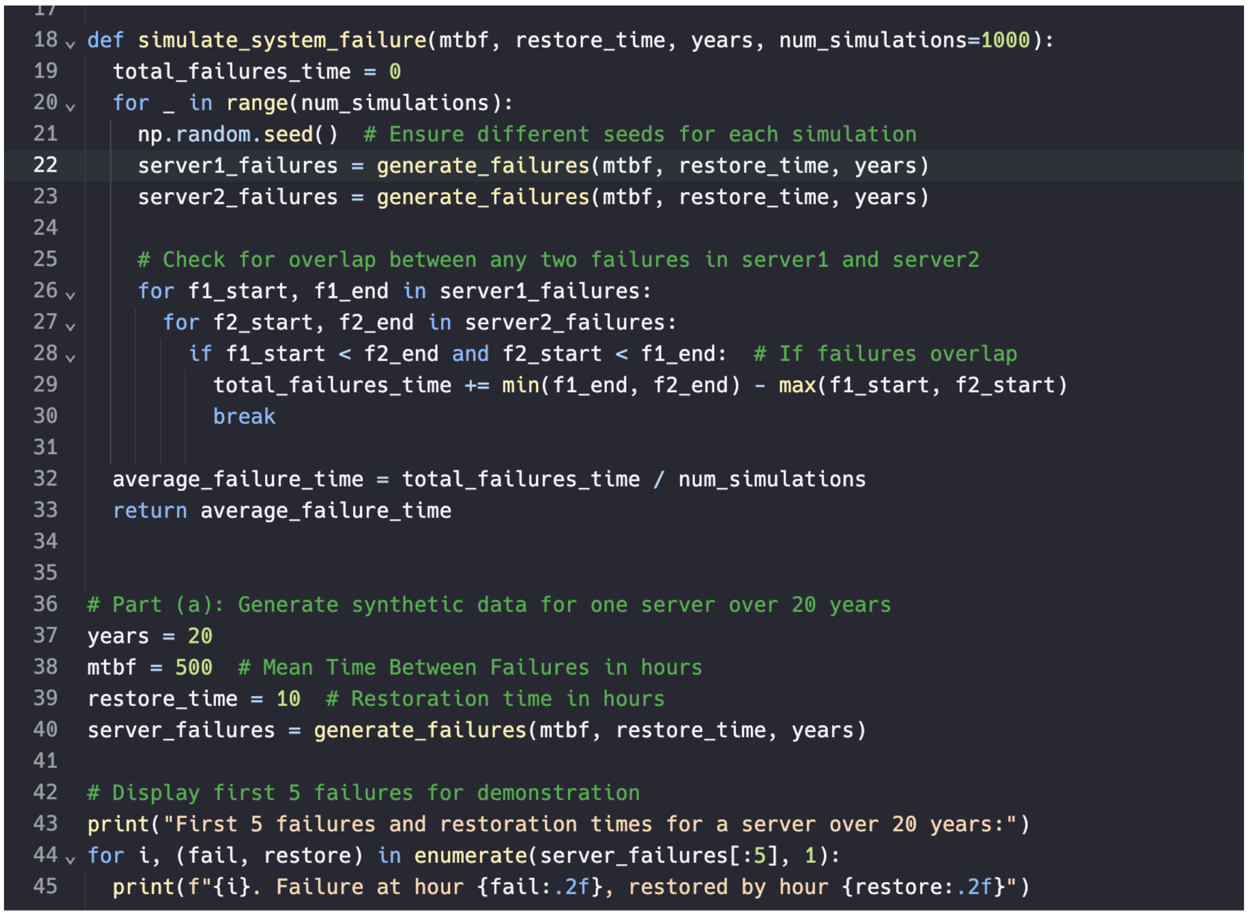
The result is a series of tuples, with each tuple capturing a failure event and the time of successful recovery.

B

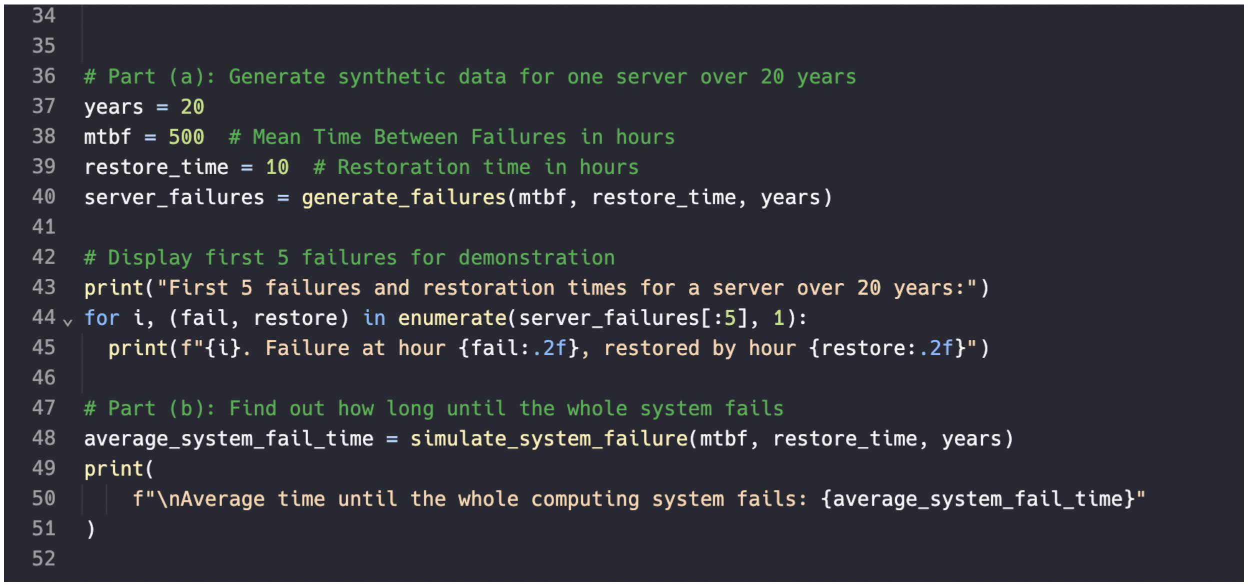
In Part B, the simulation expands to include two mirrored servers, aiming to calculate the average duration until the entire computing infrastructure encounters a failure.

A system failure is defined as occurring when both servers experience a breakdown that coincides within the 10-hour window allocated for restoration.

This simulation is executed repeatedly, each time with a unique random seed, to determine the mean time until system failure. This process offers a deeper understanding of the system's collective dependability.



```python
def simulate_system_failure(mtbf, restore_time, years, num_simulations=1000):
  total_failures_time = 0
  for _ in range(num_simulations):
    np.random.seed()  # Ensure different seeds for each simulation
    server1_failures = generate_failures(mtbf, restore_time, years)
    server2_failures = generate_failures(mtbf, restore_time, years)

    # Check for overlap between any two failures in server1 and server2
    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:  # If failures overlap
          total_failures_time += min(f1_end, f2_end) - max(f1_start, f2_start)
          break

  average_failure_time = total_failures_time / num_simulations
  return average_failure_time


# Part (a): Generate synthetic data for one server over 20 years
years = 20
mtbf = 500  # Mean Time Between Failures in hours
restore_time = 10  # Restoration time in hours
server_failures = generate_failures(mtbf, restore_time, years)

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



```python


# Part (a): Generate synthetic data for one server over 20 years
years = 20
mtbf = 500  # Mean Time Between Failures in hours
restore_time = 10  # Restoration time in hours
server_failures = generate_failures(mtbf, restore_time, years)

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

```python

# Part (b): Find out how long until the whole system fails
average_system_fail_time = simulate_system_failure(mtbf, restore_time, years)
print(
    f"\nAverage time until the whole computing system fails: {average_system_fail_time}"
)

```

The `simulate_system_failure` function models failure events for both servers across the identical 20-year timeframe, conducting numerous iterations of the simulation (typically 1000 runs).

It scrutinizes the timelines of both servers for any coinciding failure periods. The detection of such overlaps signifies a system-wide failure, and the extent of these overlaps is documented.

By averaging the time to system failure over all conducted simulations, this function estimates the system's robustness in the face of concurrent server breakdowns.