Daniel Losada and Sergio Quintanilla

In [1]:
```
import numpy as np
import itertools
import random
import scipy.stats as stats
```

1. Random permutations

1.a)

Is not fair because that would imply that all permutations should have equal probability. In this case, the choice of the second dependes on the first and Alice's positions sometimes is forced. So the distribution is not uniform.

1.b)

In [ ]:
```
def generate_fair_schedule():
    people = ['Alice', 'Bob', 'Charly', 'Dave']
    valid_permutations = []

    # Generate all possible permutations
    for perm in itertools.permutations(people):
        # We just save the ones that make Alice be in one of the first 2 turns.
        if perm.index('Alice') in [0, 1]:
            valid_permutations.append(perm)

    # Select one permutation uniformly at random from the valid ones
    return random.choice(valid_permutations)

# As we can see, now all valid permutations have equal probability of being selected, making the system fair and fulfilling th
schedule = generate_fair_schedule()
print("Bathroom schedule:", schedule)
```

```
Bathroom schedule: ('Alice', 'Bob', 'Charly', 'Dave')
```

2. Compound Poisson process

In [ ]:
```
def simulate_poisson_processes(num_simulations=10000, purchase_threshold=100):
    total_times = []

    for _ in range(num_simulations):
        # We initialize the time at 1 so we avoid division by 0
        time = 1
        # Initialize counter of purchases
        purchases = 0

        # We define when to stop iterating
        while purchases < purchase_threshold:
            # Define the compound rates
            if time < 10:
                # Initially increases with time
                rate = time
            else:
                # After time 10 the rate is static
                rate = 10

            interarrival_time = np.random.exponential(1 / rate)
            time += interarrival_time
            purchases += 1

        # Append the time taken to get the desired purchases
        total_times.append(time)

    # Return the mean time taken
    return np.mean(total_times)

def expected_time_to_sell_target(target_value=25000):
    product_prices = [200, 300, 500]
    probabilities = [0.5, 0.3, 0.2]
    expected_price_per_purchase = sum(p * prob for p, prob in zip(product_prices, probabilities))

    expected_purchases_needed = target_value / expected_price_per_purchase
    expected_time = expected_purchases_needed / 10  # Since after 10 days, λ=10

    return expected_time
```

```
# Simulate Poisson processes
average_time = simulate_poisson_processes()
print("Average time for 100 purchases:", average_time)

# Compute expected time to reach target sales
expected_sales_time = expected_time_to_sell_target()
print("Expected time to reach 25000 euros in sales:", expected_sales_time)
```

```
Average time for 100 purchases: 15.346338602341675
Expected time to reach 25000 euros in sales: 8.620689655172415
```

3. Pricing European Options

In [8]:

```
S0 = 100
K = 100
r = 0.02
sigma = 0.25
T = 1
MC = 10000

# simulate prices
np.random.seed(100)
Z = np.random.normal(0, 1, MC)  #get our Z values
ST = S0 * np.exp((r - 0.5 * sigma**2) * T + sigma * np.sqrt(T) * Z)  #plug them into the simulation formula

# future payoffs
payoffs = np.maximum(ST - K, 0)  #we pick the maximum between the difference of a strike price sample and 0, so only the value

# adjusted to today's price
adjusted_payoffs = np.exp(-r * T) * payoffs

# option price derived from the mean of adjusted payoffs
option_price = np.mean(adjusted_payoffs)
option_price
```

Out[8]:

```
np.float64(11.002300400554606)
```

In [9]:

```
# 95% confidence interval
std_error = np.std(adjusted_payoffs, ddof=1) / np.sqrt(MC)
confidence_interval = (
    option_price - 1.96 * std_error,
    option_price + 1.96 * std_error
)
confidence_interval[0],confidence_interval[1]
```

Out[9]:

```
(np.float64(10.653382692149505), np.float64(11.351218108959708))
```

Our option price comes out to around €11, with a 95% CI for €10.65 to €11.35.