1 Longest run of heads in a sequence of coin tosses

A coin is tossed n times, thereby generating a sequence of heads (H) and tails (T). A subsequence of consecutive heads is called a run of heads. Consequently, the following sequence:

HHTHTTHHHHTTTHHTTTT

of length $n = 20$ contains 5 runs of heads with lengths $2, 1, 4, 2, 1$, respectively. The longest run of heads is $M_n = 4$.

1.1 Write an efficient function (in R or Python) which generates random sequences of heads and tails for any value of n

I have called the function coinToss(n), and have chosen python for its implementation. The function receives the number of tosses (n) as argument, and returns a list with n heads (H) and tails (T), randomly distributed.

The idea to implement the function is the following. I first create an empty list that will contain the coin toss sequence. In order to generate each toss randomly I make use of the function *random.randint()*, which generates a random number between 0 and 1. If number generated is 0, tails (T) is added to the sequence, while if the random number is 1, heads (H) is added. Once the n repetitions have been generated, the function returns the random toss sequence list.

```
# Code implemented in Python

import numpy as np
import random

def coinToss (n):
    sequence=[]
    for i in range (n):
        toss=random.randint(0,1) # We generate random numbers between 0 and 1
        if toss==0:
            sequence.append("T") # If number is 0, Tails is added to the sequence
        if toss==1:
            sequence.append("H") # If number is 1, Heads is added to the sequence
    return sequence

random_toss = coinToss (10)
print (random_toss)
```

1.2 Write an efficient function which computes the length of the longest run of H in an arbitrary sequence of H and T

In this case, the function is called headRun(random_toss). The function receives a list with a random sequence of heads (H) and tails (T) as argument (generated with the coinToss(n) function), and returns the longest run of heads in the list.

The function makes use of two variables: *current_headRun*, which counts the current run of heads that is being examined at the moment, and *max_headRun*, which counts the longest run of heads found so far in the *random_toss* sequence list.

The idea of the function is to iterate through every element of the random toss sequence, and to add every consecutive H that we find to the counter of current head run. If the current head run is larger than the max, we have a new max. If we don't find an H, effectively breaking the current run of heads, *current_headRun* resets back to 0. This process is repeated for every element of the random toss sequence, and once finished the function returns the largest run of heads found.

```
def headRun (random_toss):   # Code implemented in Python
    current_headRun=0
    max_headRun=0
    for i in random_toss: # We iterate through every element of the random toss sequence
        if i=='H':
            current_headRun+=1 # Every H found is added to counter of current headRun
            if current_headRun>max_headRun:  # If the current head run larger than max:
                max_headRun=current_headRun  # we set a new max
        else:
            current_headRun=0 # If no H is found, current head run resets back to 0
    return max_headRun

random_toss = coinToss (1000)
max_headRun = headRun (random_toss)
print (random_toss)
print (max_headRun)
```

1.3 For a sequence of length $n = 1000$, we have observed a longest run of heads equal to $M_n = 6$. Basing your decision on this piece of information, is the coin fair or not?

In order to analyze whether or not obtaining a longest run of heads equal to 6 over a sequence of 1000 tosses, we need to conduct an hypothesis test. We will start supposing that obtaining a longest run of heads of M=6 can be considered as a normal outcome (null hypothesis) and, based on the probability distribution of the process, we will test whether we have statistical

proof to demonstrate the opposite, that indeed obtaining a longest run of heads of 6 is an abnormal outcome (alternative hypothesis).

It is possible to divide the process of hypothesis testing in 4 steps:

1. Formulate null and alternative hypothesis
2. Calculate t-statistic and p-value of the test
3. Establish a confidence interval
4. Test the null hypothesis

The process and considerations followed for each of these four steps have been the following:

1. **Formulate null and alternative hypothesis**

   As previously explained, for our purposes we want to consider as null hypothesis that a longest run of heads of M=6 can be considered as a normal outcome, and hence the alternative hypothesis will be that obtaining a longest run of heads of 6 is an abnormal outcome:

   - H0: $\mu = 6$
   - H1: $\mu > 6$

2. **Calculate test statistic and p-value**

   This is the most crucial part of the process. Here we want to calculate how extreme is the outcome of obtaining longest run of heads of M=6, among all the set of possible outcomes. Considering our particular case, we have two approaches:

   - **Statistical Approach.** The statistical approach aims to measure how extreme is the outcome of obtaining longest run of heads of M=6, by calculating the test statistic of the outcome based on information about the mean ($\bar{X}$) and variance (S_x) of the probability distribution:

   $$M = \bar{X} \pm t\, S_x \; ; \quad t = \frac{\bar{X} - 6}{S_x} \tag{1}$$

   Once we know the test statistic (how many standard deviations away from the mean our outcome is located), it will be possible to obtain the p-value (% of cases more extreme that our outcome) from probability distribution tables.

   - **Numerical Approach.** In a similar way to the statistical approach, the numerical approach also aims to measure the p-value of the test, but in this case it won't assume that the population follows a specific probability distribution, and will hence calculate the p-value as the cumulative frequency of obtaining M=6 as longest run of heads:

   $$\begin{aligned} \text{p-value} = cfd(6) = \text{P(longest run of heads is M=1)} + \text{P(M=2)} + ... \\ ... + \text{P(M=5)} + \text{P(M=6)} \end{aligned} \tag{2}$$

What approach should we use for our purposes? The main constraint that we face is the probability distribution of the maximum number of runs of heads within 1000 coin tosses. First of all, we don't know the probability distribution, so we would need to either i) assume that it is normal (calculate Z-statistic), or ii) assume that it is close to normal (calculate students t-statistic), or iii) run a Monte Carlo simulation to know which probability distribution our process follows.

Given that we have no basis to assume that the probability distribution of the maximum number of runs of heads within 1000 coin tosses is normal, or close to normal, we will hence take the third path and run a Monte Carlo analysis with 1000 simulations, in order to create a probability distribution of our event of study. The code for the Monte Carlo simulation has been implemented in Python:

```python
# Code implemented in Python

import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
from matplotlib import pyplot as plt

# Monte-Carlo Simulation
n_simulations=1000
max_headRun_list = []
for i in range(n_simulations):
    random_toss = coinToss (1000) # We generate 1000 random coin tosses
    max_headRun = headRun (random_toss) # Count the max number of run-heads
    max_headRun_list.append(max_headRun) # Add the max_headRun to a list

# Histogram - Create histogram with all head-runs saved in previous list
n_bins=13
x, y, patches = plt.hist(max_headRun_list, density=True,
                bins=n_bins,color='#0504aa',edgecolor="black")
plt.xticks([(patch._x0 + patch._x1)/2 for patch in patches],
           [i for i in range(6,19)])
plt.grid(axis='y', alpha=0.5)
plt.ylabel('Frequency')
plt.xlabel('Maximum Number of Head Runs per 1000 coin tosses')
plt.title('Maximum Number of Head Runs in 1000 tosses (1000 simulations)\n')

# Cumulative Distribution Function
dx = y[1] - y[0] # Calculate x-axis interval
F1 = np.cumsum(x)*dx*0.3 # Calculate cumulative frequency function (0 to 0.3)
plt.plot(y[1:], F1,color='#DC143C')
```

The results for the montecarlo simulation are shown in the histogram of Figure 1. We

can observe in blue the relative frequency of obtaining each number of maximum runs of heads within 1000 coin tosses, while the red line represents the cumulative frequency distribution (cdf) function (the y-axis for the red line goes from 0 to 100%).

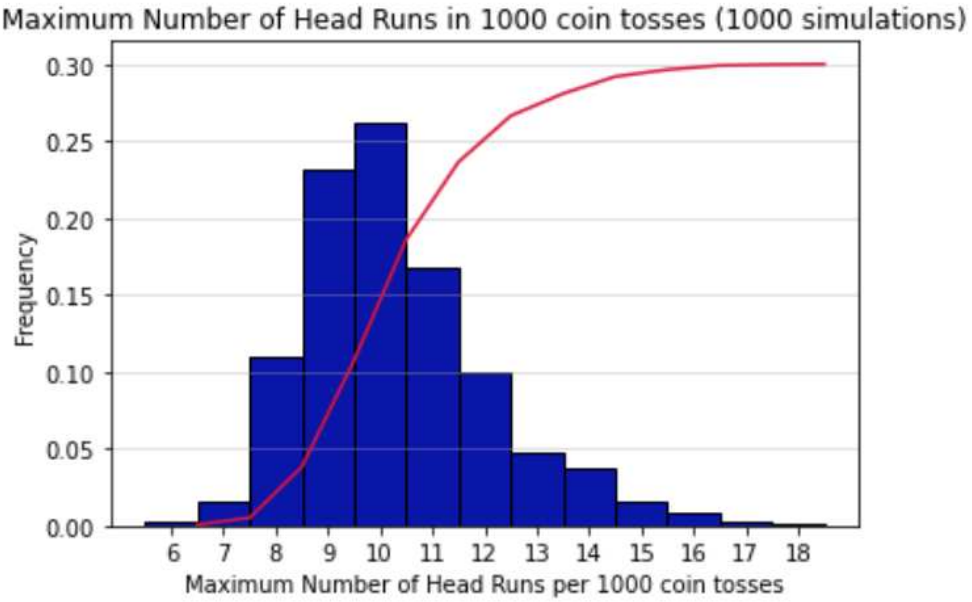

Figure 1: Histogram with Maximum Number of Head Runs in 1000 coin tosses (for 1000 Monte-Carlo simulations)

According to the results of the Monte-Carlo simulation, we can suppose that our probability distribution is log-normal (or close to log-normal). Hence we can again study the implementation of the statistical approach, or the numerical approach. Even though the ideal procedure would be to implement both approaches and compare the results, we have the inconvenience that the log-normal distribution doesn't have a procedure to calculate the test statistic (and the p-value) as straight-forward as the one shown in Equation (1) for the normal distribution.

Even though there are some approaches to calculate the test statistic and p-value for log-normal distributions (see generalized p-value or permutation methods proposed by Bhaumik et al. [1]), we will preferably focus on the numerical approach, and will make use of the Monte-Carlo simulation already generated in order to calculate the p-value, as per Equation (2). In our case, we obtain a p-value of:

$$\text{p-value} = cfd(6) = \text{P}(\text{M} \leq 6) = 0.017 \tag{3}$$

```
    # Code implemented in Python
    # Loop to compute cumulative frequency of 6 (sum of frequencies from 0 to 6)
    for i in range (min(max_headRun_list),7):
        p_value += round(max_headRun_list.count(i)/n_simulations,3)
```

3. **Establish a confidence interval**

   Once the p-value for the test has been calculated, it is time to select a confidence interval that we consider to be statistically representative of what we consider to be an "abnormal" result. For our purposes, a classic confidence interval of 99% seems appropriate. For this level of confidence, it is possible to obtain a significance level of:

$$\alpha = 1 \text{ - Confidence Interval} = 1 - 0.99 = 0.01. \tag{4}$$

4. **Test the null hypothesis**

   As commented before, it is only statistically appropriate to reject the null hypothesis if the test statistic is more extreme that our interval of confidence; or in other words, we can only reject the null if the p-value is lower than the significance level. After having calculated both, we obtain that:

$$\text{H0 CANNOT BE REJECTED: p-value (0.017)} > \text{significance level (0.01)} \tag{5}$$

```
# Code implemented in Python
print ("Hypothesis: \n H0: mu = 6 \n H1: mu < 6")
print ("Significance Level: ",alpha," (99% Confidence Interval)")
print ("p-value (Cumulative Frequency of 6): ",p_value)
if p_value < alpha:
    print ("H0 can be rejected: ",p_value," < ",alpha)
    print ("We can conclude that the coin IS NOT fair")
else:
    print ("H0 CANNOT be rejected: ",p_value," > ",alpha)
    print ("")
    print ("We can conclude that the coin is fair")
```

Thus, it is not statistically possible to reject the null hypothesis under the conditions considered, and hence we can conclude that the coin is fair.