CS 237 Lab Four: Simulating Random Variables

In this lab we will explore the notion of random variables, which will be a very important topic for the next few weeks. We will think about creating particular random variables corresponding to the various canonical problems (such as flipping a coin until a heads appears), and then think about how we can simulate particular random variables with well-known characteristics; finally we will simulate playing a game and finding the best strategy.

Lab 04 Instructions

You will use code from the following cell to implement the code in this lab. The random library has not been imported, and you may not use any libraries to implement the random functions in this lab as the whole point is for you to implement these basic functions yourself from first principles!
In [86]: # Here are some imports which will be used in code that we write for CS 237

# Jupyter notebook specific

from IPython.display import Image
from IPython.core.display import HTML
from IPython.display import display_html
from IPython.display import display
from IPython.display import Math
from IPython.display import Latex

# Imports potentially used for this lab

import matplotlib.pyplot as plt  # normal plotting
from math import log
from collections import Counter

%matplotlib inline

# Useful code from lab 01

def show_distribution(outcomes, title='Probability Distribution'):
    num_trials = len(outcomes)
    X = range(int(min(outcomes)), int(max(outcomes))+1)
    freqs = Counter(outcomes)
    Y = [freqs[i]/num_trials for i in X]
    plt.bar(X, Y, width=1.0, edgecolor='black')
    if (X[-1] - X[0] < 30):
        ticks = range(X[0], X[-1]+1)
        plt.xticks(ticks, ticks)
    plt.xlabel("Outcomes")
    plt.ylabel("Probability")
    plt.title(title)
    plt.show()

# This function takes a list of outcomes and a list of probabilities and
# draws a chart of the probability distribution.

def draw_distribution(Rx, fx, title='Probability Distribution for X'):
    plt.bar(Rx, fx, width=1.0, edgecolor='black')
    plt.ylabel("Probability")
    plt.xlabel("Outcomes")
    if (Rx[-1] - Rx[0] < 30):
        ticks = range(Rx[0], Rx[-1]+1)
        plt.xticks(ticks, ticks)
    plt.title(title)
    plt.show()

def round4(x):
    return round(x+0.000000000001,4)

def round4_list(L):
    return [round4(x) for x in L]
Problem One: Generating Random Floating-Point Numbers in [0..1)

In this problem we will investigate how to implement the function `random.uniform()`, which generates random 32-bit floating-point numbers in the range [0..1). (This is the same as `random.random()`. Essentially, this is a random variable implemented in Python. This will form the basis for a variety of similar random variables representing other canonical problems, such as flipping coins.

As you may recall from CS 112, hash functions map key values to buckets in a hash table: the hash function appears to be spreading the keys uniformly randomly over the buckets, but in fact there is nothing random about it, since we can easily repeat the computation to find the key later. This is called pseudo-random behavior: the hash function is not random, but appears to be so unless you know the rule used to compute the hash function.

The simplest hash functions use the linear-congruential method, which you may remember from CS 112; using prime numbers as multiplier and modulus are a good way to simulate random behavior. The particular choices we will use here are from this paper. For a fuller treatment, here is a nice set of lecture slides on random number generators.

```python
In [87]:
a = 914334
m = 2**22 - 3
# a = 3
# m = 7
def hash(x):
    return (a * x) % m

# Test it!
x = [231, 45, 123, 87, 133, 123]
for x in x:
    print(hash(x))
```

1496104
3396321
3411256
4049640
4165994
3411256

Pseudo-random number generation (done for you!).

However, we want to generate a series of numbers which appear to be uniformly randomly distributed over the range [0 .. m), using a Pseudo-random number generator (see the excellent Wikipedia article for more details), and so we will start with a seed value and successively apply the hash function to generate a series of pseudo-random numbers $n_1, n_2, n_3, \ldots$.

Supposing that our initial "seed" value is 1, we would have:

\[
\begin{align*}
n_0 &= 1 \\
n_1 &= \text{hash}(1) \\
n_2 &= \text{hash}(\text{hash}(1)) \\
& \vdots \\
n_k &= \text{hash}^k(1)
\end{align*}
\]
Part (a): Your Turn! Pseudo-random Floats.

Now convert this into a random variable which produces floating-point values in the range \([0..1)\). This is simply our own version of the Python function `random.random()`.

Hint: What would you divide the large random number by?

```python
In [89]: # Solution (a) -- Creating our own version of the Python function random.random()
    
0.6634089923446124
0.39760641880494507
0.06733160066480684
0.571762255498592
0.6701190496342537
0.6311282857381957
0.05001214735293745
0.8068131018732323
0.6506881599580001
0.3080470381119524
```
Testing for Randomness: Test One -- The Pair Test.

Now we will test our function `my_uniform()`. For the first test, simply run the next cell, which will display a sequence of random points in a plane bounded by [0..1) in the X and Y axis. You should see a randomly spread out collection of points with no discernable patterns.

All you have to do is to generate `num_trials` random variates in two lists; as we have seen in previous labs, `matplotlib`'s plotting functions take the x values and the y values in two separate lists, that is, the points

\[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\]

are represented by the lists

\[X = [x_1, x_2, \ldots, x_n]\]
\[Y = [y_1, y_2, \ldots, y_n]\]

```
In [90]: my_seed(0)
```
Part (b): Testing my_random() for Randomness: Test Two -- The Period Test

There are two important characteristics of pseudo-random number generators: how long a sequence of pseudo-random numbers they can produce, and how well they spread the numbers equinumerously over their range.

The period of a generator is how long it takes before it starts to repeat the pseudo-random sequence, i.e., the number of unique values produced. Why does it repeat? Well, consider the case when \( a = 3 \) and \( m = 7 \) with an initial seed of 0:

\[
\begin{align*}
n_0 &= 3 \\
n_1 &= 2 \\
n_2 &= 6 \\
n_3 &= 4 \\
n_4 &= 5 \\
n_5 &= 1 \\
n_6 &= 3 \\
n_7 &= 2 \\
n_8 &= 6 \\
\end{align*}
\]

You can see that when we reach \( n_6 \), the sequence starts to repeat, at which point the sequence will be exactly the same again (why?). The period of this generator is \( m - 1 = 6 \). No linear-congruential generator in the form we have given it can have a period larger than \( m - 1 \) (why?), so this choice of \( a \) and \( m \) gives us a full-period generator. Not all generators will have the full period, but we want as long a period as possible given \( m \), since we do not want pseudo-random numbers to repeat before the end of our random experiment--this would pretty much be the opposite of random behavior!

Todo: For the value of \( m \) and \( a \) we are using, determine the period by generating successive values of \( n_k \) until you get a duplicate value. Answer this question: Does this choice of \( a \) and \( m \) lead to a full-period generator?

Hint: Try setting \( a \) to 3 and \( m \) to 7 above and seeing if the period is indeed 6, and then set it back to the original values to answer the question.

In [91]: # Solution (b)

    Period is 4194300
    Yes, this is a full-period generator.

Part (c): Testing my_random() for Randomness: Test Three -- The Spectral Test.

For this part, we would like you to do the spectral test. In order to do this, since these are very close to behaving like real numbers, as with continuous probability we will need to *bin* the numbers into some suitable intervals. For any number \( k \) of equal-width bins over the range, we would expect the probability of a number landing in a particular bin to be very close to \( \frac{1}{k} \).

There is a subtle problem with this test and its interaction with the period: if you generate exactly as many numbers as in the period, then by definition the bins will be almost exactly the same size, since the sequence generated all possible numbers in the range \([0 .. m)\). (You can verify this by doing \( m = 4194301 \) trials.) Hence we do not want to generate too much of the sequence to do the spectral test. We will therefore use only \( 10^5 \) numbers, which represent

\[
\frac{10^5}{4194301} = 0.0238
\]

or about 2% of the period.

Todo:

Generate \( 10^5 \) values using my_uniform().

Convert these floating-point numbers into integers in the range \([0, ..., 100)\) by multiplying by 100 and then converting to an int (which will truncate the fractional part). If we histogram the sequence, we should get an approximately equinumerous distribution over the range \([0, .., 100)\).

The code for doing the bar chart was developed in Lab 01, and the solution is included in the first code cell above.
Problem Two: Generating Random Floats and Integers in a Range

Now we will investigate generating random floats and integers in a specific range, simulating the functions `uniform(...)` and `randint(...)` respectively.

In order to transform a float in the range $[0..1)$ to the new range $[a..b)$, simply take the random variate from $[0..1)$, rescale it to the range $[0..(b-a))$, and then shift it by adding $a$.

Then, to create integers in the same range, we will simply convert the floats produced by this function to integers. Technically, this will not work for `my_randint(...)` if $a$ and $b$ are not integers, but we will assume this is the case.

We will test only the second function, since it automatically bins the floats into integer bins.
Problem Three: Choosing, Shuffling, and Sampling from a List

Now we will create our own versions of the `sample(...)`, `shuffle(...)`, and `choice(...)` functions, which we used as a basis for the code in Lab 03.

All you need to do is to demonstrate these as shown. We could test them using probability distributions, but it will suffice to check the results by eye...

Part (a): Choosing from a list with replacement

This part is easy: to choose a member of a list randomly and with replacement, simply generate a random integer as an index and return the member at the index.

```python
In [94]: # Return a list of length size of elements from the list X; the default for size is 1

[1, 1, 1, 1, 1, 1, 1, 3]
[2, 4, 2, 1, 2, 6, 5, 1]
[3, 5, 5, 1, 2, 6, 4, 1]
[1, 4, 6, 4, 4, 4, 6, 2]
[3, 2, 2, 1, 2, 4, 2, 1]
[4, 5, 6, 4, 2, 2, 6, 1]
[3, 1, 4, 5, 1, 2, 6, 6]
[1, 5, 4, 5, 1, 1, 1, 4]
[1, 1, 5, 6, 5, 3, 3]
[6, 2, 6, 6, 3, 6, 2, 2]
```

Part (b): Shuffling a list

This is exactly the same as shuffling a deck and then dealing out a number of cards from the top. In order to do this, we shall use the following method, known as the Fisher-Yates Shuffle (https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle), which works in $O(n)$.

Basically, you maintain a shuffled part of the list and an unshuffled part of the list, and at each step randomly select a number from the unshuffled part and move it to the shuffled part.

```
-- To shuffle an array A of n elements (indices 0...n-1):
for i from n-1 downto 1 do
    j ← random integer such that 0 ≤ j ≤ i
    exchange A[j] and A[i]
```

Since this shuffle works in-place, you should create a copy of the list before shuffling it.

```python
In [95]: # Shuffle using the Fisher-Yates algorithm.

[3, 4, 2, 5, 6, 7, 8, 9, 10, 1]
[4, 3, 10, 8, 5, 6, 9, 1, 2, 7]
[6, 3, 8, 5, 10, 4, 7, 2, 1, 9]
[10, 8, 4, 1, 3, 2, 7, 9, 5, 6]
[3, 7, 8, 5, 9, 10, 1, 2, 6, 4]
[10, 9, 7, 6, 5, 1, 4, 8, 2, 3]
[9, 3, 6, 1, 7, 4, 5, 2, 8, 10]
[4, 9, 8, 3, 10, 5, 6, 2, 1, 7]
[6, 4, 1, 10, 5, 3, 7, 8, 2, 9]
[10, 5, 4, 1, 3, 9, 7, 8, 6, 2]
```

Part (c) Sampling without replacement from a list

This is easy: just slice the list produced by shuffling.
Problem Four: Generating Random Variates by Simulation

A random variate is a number produced by a random number generator following some probability distribution. Basically, you are implementing a particular random variable which you can "poke" to get random variates.

In this problem we will consider a simple way of generating such variates by simply simulating the canonical problem associated with a particular distribution.

We will start to look at the distributions mentioned here at the next lecture.

Part (a): Generating the Bernoulli Distribution

In this problem we will investigate how to implement a random variable that simulates the flipping of a (possibly unfair) coin, where the probability of a heads is \( p \), and returning 1.0 ("success") if a heads turns up, and 0.0 ("failure") if tails; the random variable is thus:

\[
S = \{ 0.0, 1.0 \} \\
\text{P} = \{ 1-p, p \}
\]

In [97]:

```
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
0.0
```

In [98]:

![Probability Distribution for Bern(0.7)](image)
Part (b): Generating a Binomial Random Variable

In this problem we will investigate how to implement a random variable that simulates the following problem: Flip a (possibly unfair) coin (where the probability of heads is $p$) $n$ times -- how many heads appeared?

We shall generate these random numbers by simulating the flipping of coins, using the solution to the previous problem.

In [99]:

In [100]:
Problem Five: Let's Gamble!

Now suppose we want to actually figure out the right way to gamble, using a computer simulation. We’ll go back to dice....

We will play a version of the game "Twenty One" and figure out the best strategy.

This game basically a one-player version of the famous card game Blackjack. The game is played for some number \( n \) of rounds (we will use \( n = 10^5 \)), at the end of which the player wins points. The player accumulates points during the whole game, and of course whoever gets the most points wins.

The objective in each round of the game is to score as close to 21 as possible by rolling a die as many times as you wish and adding all the numbers that appear. When a player’s total exceeds 21, he is ‘busted’ and gets 0 points. If the player chooses to stop rolling before he exceeds 21, then the sum of the numbers rolled in that round is the number of points he wins. (There are many variations on this game, some involving multiple players, or a “banker” or different numbers of dice, or alcohol….. the most famous version is Blackjack and you can check out YouTube for details.)

A computer can play this game with a suitable strategy. For this problem, we will consider a strategy to be simply an integer \( k \) which is the value at which you stop rolling (thinking that you are close enough to 21). The number \( k \) is fixed for the entire game. For example, if you set \( k = 19 \), then in every round, you will keep rolling if your sum to that point is less than or equal to 19; if you get more than 19 you stop.

You must write a function \( \text{play\_round}(k) \) which rolls a single die until you reach \( k \) or get busted, and either return your score (if you reached \( k \)), or 0 (if you were busted). Then you will write a function \( \text{play\_game()} \) which calls \( \text{play\_round}(k) \) for \( n = 10^5 \) times for each \( k \) and returns an array of 21 numbers giving the average payoff for each \( k \).

Finally,

- Print out a suitable bar chart of the results, noting that this is NOT a probability distribution, but a bar chart of average payoff (on the Y axis) vs. strategy (on the X axis). Modify the code from the first cell to print out an attractive chart with appropriate labels.
- Finally, answer the following question: What is the best strategy for the game, meaning what value of \( k \) wins the most points on average?

In [102]:

```
```

Best score is obtained for \( k = 16 \)