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 random variables for certain characteristic frequency distributions corresponding to the various canonical problems (such as flipping a coin until a heads appears), and then explore the notion of expected value and the best strategy for a game.

In [432]:

```python
# 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 numpy as np           # arrays and functions which operate on array
import matplotlib.pyplot as plt       # normal plotting
import seaborn as sns           # Fancy plotting
import pandas as pd             # Data input and manipulation

from numpy.random import random, randint, uniform, choice, shuffle
from collections import Counter

%matplotlib inline
```

Problem One: Generating Random Floating-Point Numbers in [0..1]

In this problem we will investigate how to implement the function `numpy.random.random()`, which generates random 32-bit floating-point numbers in the range [0..1). 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.

Hash functions 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:
Part (A): Pseudo-random number generation.

However, we want to generate a series of numbers which appear to be uniformly randomly distributed over the range \([0 .. M)\), and so we will start with a seed value and successively apply the hash function to generate a series of pseudo-random numbers \(n_0, n_1, n_2, \ldots\).

\[
\begin{align*}
n_0 &= \text{hash}(\text{seed}) \\
n_1 &= \text{hash}(\text{hash}(\text{seed})) \\
\vdots \\
n_k &= \text{hash}^{k+1}(\text{seed})
\end{align*}
\]

Part (B): Pseudo-random Floats.

Now convert this into a random variable which produces floating-point values in the range \([0..1)\):
Part (C) Test Two: The Histogram Test.

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 a sequence of 1 million such values, we should get an approximately uniform distribution over the range \([0, .., 100)\).

```python
In [435]:
1     def nextUniform():
2         pass
3
4     # Test it!
5     for x in range(1,11):
6         print(nextUniform())
```

```python
0.7827998493886192
0.5987375228122295
0.4242822084192676
0.12388548345187628
0.4728314585063862
0.4310389429412462
0.1310943277531439
0.9225541595175065
0.3297236734355619
0.5441774181634972
```

Problem Two (A): Generating Random Integers in a Range \([a, ..., b)\)

Now we will investigate generating random integers in a specific range, from \(a\) (inclusive) to \(b\) (exclusive, as usual in ranges in Python); this is equivalent to the numpy.random function `randint(a,b)`; the random variable looks like this:

\[
S = \{ a, a+1, ..., b-1 \}
\]

\[
P = \{ 1/(b-a), 1/(b-a), ..., 1/(b-a) \}
\]

As we shall explore later this week, this is called a Discrete Uniform random variable.
Problem Two (B): Sampling without replacement

In this problem we will create our own version of the numpy choice function, which we used extensively in Lab Three. There are two steps: first create a function shuffle(...) which takes a list and creates a random permutation, and then slice the list to return some number size of elements from the front of the list. It is exactly the same as shuffling a deck and then dealing out a number of cards from the top.

```python
In [438]:
def nextDiscreteUniform(a,b):
    pass

# Test it!
for k in range(10):
    print(nextDiscreteUniform(0,10))
```

```python
In [461]:
num_trials = 10000
X = [nextDiscreteUniform(0,10) for k in range(num_trials)]
D = Counter(X)
K = list(D.keys())
K.sort()
P = [D[k]/num_trials for k in K]
plt.bar(K,P,width=1.0,edgecolor='k')
plt.show()

# Test this by running this cell several times!
```

![Histogram of discrete uniform distribution](attachment:image.png)
In [440]:
1 # shuffle a list X by choosing two indices using the function nextDiscreteUniform just created
2 # in part (A) and swapping the two elements at those indices. Repeat this 10 * len(X) times.
3 # Do NOT destroy the list, but make a copy before shuffling it.
4
5 def shuffle(X):
6     pass
7
8 X = [1,2,3,4,5,6,7,8,9,10]
9 for k in range(10):
10    print(shuffle(X))

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

In [462]:
1 # Return a list of length size of elements from the list X; if replace is True,
2 # simply select elements from the list using nextDiscreteUniform; if False, shuffle
3 # the list and slice an initial part of the list and return it.
4
5 def my_choice(X,replace=False,size=1):
6     pass
7
8 # Test it!
9 X = [1,2,3,4,5,6,7,8,9,10]
10 for k in range(10):
11    print(my_choice(X,replace=True,size=8))
12
13 print()
14 for k in range(10):
15    print(my_choice(X,replace=False,size=8))

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

Problem Three (A): Generating Random Coin Flips

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 counting the number of heads: the random variable is therefore:

\[
S = \{ \ 0, \ 1 \ \}
\]

\[
P = \{ \ 1-p, \ p \ \}
\]

As we shall explore later this week, this is called a Bernoulli random variable.
Problem Three (B): Counting Heads in N Random Coin Flips

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?

As we shall explore later this week, this is called a Binomial random variable.

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

```python
def nextBernoulli(p):
    pass

for k in range(10):
    print(nextBernoulli(0.6))
```

```python
num_trials = 10000
X = [int(nextBernoulli(0.6)) for k in range(num_trials)]
D = Counter(X)
K = list(D.keys())
K.sort()
P = [D[k]/num_trials for k in K]
plt.bar(K,P,height=1.0,edgecolor='k')
print(P)
```

```
[0.4013, 0.5987]
```
Problem Four: How many flips until a head appears?

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) until a head appears -- how many flips did it take?

The key thing to understand is that if it takes \( k \) flips, then it took \( (k-1) \) tails (with probability \( 1-p \)) and then one head (with probability \( p \)):

\[
S = \{1, 2, 3, \ldots, k\} \\
\mathbb{P} = \{p, (1-p)p, (1-p)^2p, \ldots, (1-p)^{k-1}p\}
\]

As we shall explore later this week, this is called a Geometric random variable.

We could do a simulation, as in the last problem, but we will explore two new techniques for generating random variables, one based on the Cumulative Distribution Function, and the other based on an explicit formula for transforming the Uniform distribution for \([0..1)\) into the random variable we are trying to create.

Part (A) Using the CDF to generate random values

The algorithm for doing this is actually very simple: just generate a random value \( U \) in the range \([0..1)\) and when you sum up the values in the CDF, stop when you exceed \( U \). For example, if \( p = 0.3 \), then the CDF is as follows:

```python
def nextBinomial(N, p):
    pass
for k in range(10):
    print(nextBinomial(5, 0.5))
```

```python
num_trials = 100000
X = [int(nextBinomial(4, 0.5)) for k in range(num_trials)]
D = Counter(X)
K = list(D.keys())
K.sort()
P = [D[k]/num_trials for k in K]
plt.bar(K, P, width=1.0, edgecolor='k')
print(P)
```
In [446]:
1 p = 0.3
2 X = range(1,20)
3 probs = [(1-p)**(k-1)*p for k in X]
4 cum_probs = np.zeros(len(probs))
5 for i in range(len(probs)):
6     for j in range(i+1):
7         cum_probs[i] += probs[j]
8
9 plt.bar(X,cum_probs, tick_label=X,width=1.0,edgecolor='black')
10 plt.title('Cumulative Distribution Function')
11 plt.ylabel("P(X<=k)"
12 plt.xlabel("k in Range(X)"
13 plt.show()
14
15

Rng(X) = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
F(X) = [0.3 0.51 0.657 0.7599 0.83193 0.882351 0.9176457 0.94235199 0.95964639
0.97175248 0.98022673 0.98615871 0.9903111 0.99321777 0.99525244
0.99667671 0.99767369 0.99837159 0.99886011]

In [447]:
1 def f(k,p):
2     pass
3
4 def nextGeometric(p):
5     pass
6
7 # Test it
8 for k in range(10):
9     print(nextGeometric(0.3))
10
11
12
13
Part (B) Using an explicit formula to generate random values

In the previous problems we considered generating random numbers by simulation and by the inverting the CDF. Now we will explore using an explicit function for the inverse of the CDF. The following formula is described in the literature: if $U$ is a value uniformly-distributed in the range $[0..1)$, then

$$1 + \text{floor}\left[ \frac{\log(U)}{\log(1-p)} \right]$$  

# log is to the base e

is an integer which is distributed according to the Geometric Distribution with probability $p$.

```python
In [448]:
1 num_trials = 100000
2 X = [int(nextGeometric(0.3)) for k in range(num_trials)]
3 D = Counter(X)
4 K = list(D.keys())
5 K.sort()
6 P = [D[k]/num_trials for k in K]
7 plt.bar(K,P,width=1.0,edgecolor='k')
8 print(P)

[0.2976, 0.20945, 0.14756, 0.10354, 0.0727, 0.05049, 0.0355, 0.02524, 0.01741, 0.01185, 0.00852, 0.00587, 0.00436, 0.0028, 0.00224, 0.00137, 0.00105, 0.00083, 0.00051, 0.00026, 0.00021, 0.00002, 9e-05, 0.0001, 7e-05, 5e-05, 4e-05, 4e-05, 1e-05, 1e-05, 1e-05, 1e-05, 1e-05]

In [457]:
1 def nextGeometric2(p):
2     pass
3
4 # Test it!
5 for k in range(10):
6     print( nextGeometric2(0.3) )
```
In [454]:
1. num_trials = 100000
2. # Display experimental distribution for Geo(0.3)
3. X = [int(nextGeometric2(0.3)) for k in range(num_trials)]
4. D = Counter(X)
5. K = list(D.keys())
6. K.sort()
7. P = [D[k]/num_trials for k in K]
8. plt.bar(K,P,width=1.0,edgecolor='k')
9. print(P)

[0.29641, 0.21157, 0.14751, 0.10207, 0.07171, 0.05167, 0.03615, 0.02493, 0.01747, 0.01237, 0.00852, 0.00596, 0.00404, 0.00293, 0.00195, 0.0014, 0.00099, 0.00079, 0.00043, 0.00026, 0.00025, 0.00016, 0.00013, 0.00014, 6e-05, 5e-05, 5e-05, 1e-05, 1e-05, 1e-05]