Question 1

(This is Problem 4.4 from Gregory.) A bottle contains three green balls and three red balls. The bottle is first shaken to mix up the balls.

(a) (2 pts) Calculate analytically the probability that blindfolded, you will pick a red ball on the third pick, if you learn that at least one red ball was picked on the first two picks.

(b) CODING: (3 pts) Write a Python function that simulates this an instance of drawing three balls from the bottle.

(c) CODING: (3 pts) Using the function from b), empirically calculate the probability from part a).

Quesiton 1a:

BEGIN SOLUTION

Define events:

```
A = pick a red ball on the third pick
B = pick a red ball on one of the first two picks
```

Then the value we want is:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

This is easier if we divide B into three possibilities:

```
B1 = pick red then green
B2 = pick green then red
B3 = pick red twice
```

These are easy to calculate:

$$\begin{align} P(B_1) &= (3/6) \times (3/5) = 3/10 \\ P(B_2) &= (3/6) \times (3/5) = 3/10 \\ P(B_3) &= (3/6) \times (2/5) = 1/5 \end{align}$$

Then,

$$P(B) = P(B_1) + P(B_2) + P(B_3) = 4/5$$

For the numerator, we again consider A coupled to each sub-possibility for B:

$$P(A \cap B_1) = (3/10) \times (2/4) = 3/20 \tag{4}$$
$$P(A \cap B_2) = (3/10) \times (2/4) = 3/20 \tag{5}$$
$$P(A \cap B_3) = (1/5) \times (1/4) = 1/20 \tag{6}$$

Then,

$$P(A \cap B) = P(A \cap B1) + P(A \cap B2) + P(A \cap B3) = 7/20$$

Finally,

$$P(A|B) = \frac{7/20}{4/5} = 7/16$$

END SOLUTION

In [1]:

```
# Question 1b

import numpy as np

def simulate_draw3():
    """Simulate drawing 3 balls from a bottle containing 3 green balls and 3 re

    Returns: an array of three strings, each of which is either 'G' or 'R',
             representing green and red respectively.
    """
    ### BEGIN SOLUTION
    balls = ['G', 'G', 'G', 'R', 'R', 'R']
    draw = np.random.choice(balls, 3, replace=False)
    return draw
    ### END SOLUTION
```

In [2]:

```
draw = simulate_draw3()

# If you run this multiple times, this should (usually) change each time.
print(f'The three draws are: {draw}')

# Some sanity checks:

# Length should be 3
assert len(draw) == 3

# Each one should be either 'G' or 'R'
d1, d2, d3 = draw
assert d1 in ['G', 'R']
assert d2 in ['G', 'R']
assert d3 in ['G', 'R']

### BEGIN HIDDEN TESTS

# This code snippet lets us just log the test failures so we can see all the th
# bomb out on the first failure.  Then at the end, we just assert that there we
# automatic grading to give points.
nfail=0
from contextlib import contextmanager
```

```
import traceback
@contextmanager
def log_assert():
    global nfail
    try:
        yield
    except AssertionError as e:
        print('Failed assert:')
        print(traceback.format_exception(e)[-2].split('\n')[1])
        print('    msg =',str(e))
        nfail += 1

# Two successive runs *might* be identical, so just check that after 10 runs, t
draws = [simulate_draw3() for i in range(10)]
same = [np.array_equal(d, draw) for d in draws]
with log_assert():
    assert not np.all(same), same

print(f"\nTotal of {nfail} test failures")
assert nfail == 0
### END HIDDEN TESTS
```

```
The three draws are: ['R' 'G' 'R']

Total of 0 test failures
```

In [3]:
```
# Question 1c

def estimate_prob():
    """Estimate the probability of drawing a red ball 3rd given that at least o
    draws were red, by runing simulate_draw3 many times.

    Returns
    """
    ### BEGIN SOLUTION

    nsims = 100_000

    draws = (simulate_draw3() for i in range(nsims))

    num = 0    # All draws with 3rd draw R, and one of first two R
    denom = 0  # All draws with one of first two R
    for d in draws:
        if d[0] == 'R' or d[1] == 'R':
            denom += 1
            if d[2] == 'R':
                num += 1
    return num / denom
    ### END SOLUTION
```

In [4]:
```
prob = estimate_prob()
print(f'The estimated probability is {prob}')

# You should compare this to what you got in Question 1a.  It should be equal t

### BEGIN HIDDEN TESTS
nfail=0
```

```
with log_assert():
    assert np.isclose(prob, 7/16, atol=0.01), (prob, 7/16)

# Doing it again should give a different answer
prob2 = estimate_prob()
with log_assert():
    assert prob2 != prob, (prob2, prob)

# And neither should be *exactly* 7/16.  This should be exceedingly unlikely!
with log_assert():
    assert prob != 7/16
with log_assert():
    assert prob2 != 7/16

print(f"\nTotal of {nfail} test failures")
assert nfail == 0
### END HIDDEN TESTS
```

```
The estimated probability is 0.44130051184501984

Total of 0 test failures
```

Question 2

(6 pts; this is Problem 4.7 from Gregory.) In a particular water sample, ten bacteria are found, of which three are of type A. Calculate analytically the probability of obtaining six type A bacteria, in a second independent water sample containing 12 bacteria in total.

Hint: You may find the following definite integral to be helpful:

$$\int_0^1 dx x^k (1-x)^{n-k} = \frac{1}{(n+1)\binom{n}{k}}$$

BEGIN SOLUTION

Our model is that a fraction X of the bacteria in the source are of type A. We assume that the prior on X, before taking any data, is uniform $0 < X < 1$.

After taking the first water sample and finding 3 of 10 bacteria are type A, the likelihood for X is:

$$P(X|D_1) = \frac{P(D_1|X)P(X)}{P(D_1)}$$

$P(D_1|X)$ follows a binomial distribution:

$$P(D_1|X) = \binom{10}{3} X^3 (1-X)^7$$

$P(D_1)$ in the denominator acts as the normalization coefficient, which is the integral of this over all possible X.

$$
\begin{aligned}
P(D_1) &= \int_0^1 dXP(D_1|X) &(7)\\
&= \binom{10}{3}\int_0^1 dXX^3(1-X)^7 &(8)\\
&= \binom{10}{3}\frac{1}{(11)\binom{10}{3}} &(9)\\
&= \frac{1}{11} &(10)
\end{aligned}
$$

Putting these together, we get

$$P(X|D_1) = 1320X^3(1-X)^7$$

This becomes our prior for the second experiment, taking a sample that has 12 bacteria. The probability of obtaining 6 type A bacteria in this sample is

$$
\begin{aligned}
P(D2|D_1) &= \int_0^1 dXP(D2|X)P(X|D1) &(11)\\
&= \int_0^1 dX\left(\binom{12}{6}X^6(1-X)^6\right)\left(1320X^3(1-X)^7\right) &(12)\\
&= \binom{12}{6}(1320)\int_0^1 dXX^9(1-X)^{13} &(13)\\
&= \binom{12}{6}(1320)\frac{1}{(23)\binom{22}{9}} &(14)\\
&= \frac{924\times 1320}{23\times 497420} &(15)\\
&= \frac{792}{7429} &(16)\\
&\approx 0.10661 &(17)
\end{aligned}
$$

END SOLUTION

Question 3

This problem demonstrates empirically several properties of counting statistics, which obeys the Poisson distribution. We will approximate a Poisson process by generating random values between 0 and 10, but only looking at the ones between 0 and 1. Technically, this is a multinomial distribution, but for large N and at least moderately large number of bins, it is a good approximation to a Poisson process.

(a) (3 pts) Generate $N = 10^6$ uniform random deviates between 0 and 10. Count the number of deviates that fall in each of 100 equal-sized bins between 0 and 1 (0.00-0.01, 0.01-0.02, etc). Calculate the mean μ and variance σ^2 of the counts per bin. Make sure that the mean is almost exactly what you expect: $\bar{n} \approx N/100 = 1000$.

(b) (3 pts) Repeat part (a) for 10 and 1000 equally-sized bins. This empirically shows that $\mu = \sigma^2 = \bar{n}$, which is an important property of the Poisson distribution.

(c) **EXTRA CREDIT** (2 pts): Now generate $M = 100$ realizations of part (a). Make sure to not use the same random seed each time. You should now have $M = 100$ values of both the mean and variance. Let's call these μ_i and σ_i^2, respectively, where $i = 1, 2, \ldots, M$. Now calculate the mean and variance of μ_i and σ_i^2. This demonstrates that the general property that the variance (2nd moment) is much more difficult to measure accurately than the mean (1st moment).

In [5]:
```
# Question 3a,b

def make_counts(ntot, nbins):
    """Make a random array of ntot uniform random deviates between 0 and 10,
    and bin the values between 0 and 1 into nbins bins.

    Returns the number of values in each bin as an array of lenth nbins
    """
    # Hints:
    # 1. Use either np.random.random or np.random.uniform to make the full arra
    # 2. There are a number of ways to bin these.  Probably np.histogram is eas
    #    Read the docs carefully, since you'll need to set some non-default par

    ### BEGIN SOLUTION

    x = np.random.uniform(0,10,ntot)
    # Make sure to give the full range so the bins are uniform over the full po
    # not min(x) to max(x).
    hist, edges = np.histogram(x, nbins, range=(0,1))

    # That's it.  This is the array we wanted.
    return hist

    ### END SOLUTION

def calculate_mean_var(x):
    """Return the mean and variance of the values in the given array (as a tupl
    """
    # You've done this twice already, so this should be a trivial one-liner at
    # Feel free to use numpy functions for this.

    ### BEGIN SOLUTION
    return np.mean(x), np.var(x)
    ### END SOLUTION
```

In [6]:
```
N = 1_000_000 # The underscores are ignored. For large integers, they are allow

# 4a:
nbins = 100

x = make_counts(N, nbins)
mean, var = calculate_mean_var(x)

print(f"Mean of {nbins} values = {mean:.2f}")
```

```
print(f"Expected mean = {N/10/nbins:.2f}")

print(f"Variance of {nbins} values = {var:.2f}")
print(f"Expected variance = {N/10/nbins:.2f}")

# 4b:
nbins = 10
x = make_counts(N, nbins)
mean, var = calculate_mean_var(x)

print()
print(f"Mean of {nbins} values = {mean:.2f}")
print(f"Expected mean = {N/10/nbins:.2f}")

print(f"Variance of {nbins} values = {var:.2f}")
print(f"Expected variance = {N/10/nbins:.2f}")

nbins = 1000
x = make_counts(N, nbins)
mean, var = calculate_mean_var(x)

print()
print(f"Mean of {nbins} values = {mean:.2f}")
print(f"Expected mean = {N/10/nbins:.2f}")

print(f"Variance of {nbins} values = {var:.2f}")
print(f"Expected variance = {N/10/nbins:.2f}")



### BEGIN HIDDEN TESTS
nfail = 0

# To TA's: The tests here are at 5 sigma, so they are pretty unlikely to fail b
# However, before investigating too closely, run the tests again.  If they pass

x = make_counts(N, 100)
mean, var = calculate_mean_var(x)
with log_assert():
    assert len(x) == 100, len(x)
with log_assert():
    # Expected uncertainty is +- 3, so this is 5 sigma check.
    assert np.isclose(mean, 1000, rtol=0.015), mean
with log_assert():
    # Expected uncertainty is +- 141, so this is 5 sigma check.
    assert np.isclose(var, 1000, rtol=0.7), var

# These aren't necessarily obvious, but:
# Relative uncertainty for mean scale as 1/sqrt(N); insensitive to nbins
# Relative uncertainties for var scale as 1/sqrt(nbins); insensitive to N

x = make_counts(N, 10)
mean, var = calculate_mean_var(x)
with log_assert():
    assert len(x) == 10, len(x)
with log_assert():
    assert np.isclose(mean, 10000, rtol=0.015), mean
```

```
with log_assert():
    assert np.isclose(var, 10000, rtol=2), var

x = make_counts(N, 1000)
mean, var = calculate_mean_var(x)
with log_assert():
    assert len(x) == 1000, len(x)
with log_assert():
    assert np.isclose(mean, 100, rtol=0.015), mean
with log_assert():
    assert np.isclose(var, 100, rtol=0.2), var

# Drive up the number of bins to make the test a bit more stringent.  Should st
mean, var = calculate_mean_var(make_counts(N*10, 10000))
#print('mean, var = ', mean, var)
with log_assert():
    assert np.isclose(mean, 100, rtol=0.005), mean
with log_assert():
    assert np.isclose(var, 100, rtol=0.07), var

# Check some low values of N to make sure they actually span the desired range.
x1 = make_counts(10,10)
with log_assert():
    assert len(x1) == 10, len(x1)
# Should on average only have 1 bin with non-zero value.  Sometimes 2 or 3, but
#print('x = ',x)
# I didn't calculate the probability that this could fail.  But I think it must
with log_assert():
    assert np.sum(x1==0) > np.sum(x1==1), (np.sum(x1==0), np.sum(x1==1))

print(f"\nTotal of {nfail} test failures")
with log_assert():
    assert nfail == 0
### END HIDDEN TESTS
```

```
Mean of 100 values = 1000.30
Expected mean = 1000.00
Variance of 100 values = 1023.91
Expected variance = 1000.00

Mean of 10 values = 10032.20
Expected mean = 10000.00
Variance of 10 values = 8658.16
Expected variance = 10000.00

Mean of 1000 values = 99.85
Expected mean = 100.00
Variance of 1000 values = 100.83
Expected variance = 100.00

Total of 0 test failures
```

In [7]:

```
# Problem 3c:

def run_simulation(M, N, nbins):
    """Run M realizations of the calculation done for Problem 4a

    Returns two lists (mu, sigsq), each of length M.
```

```
    """
    # BEGIN SOLUTION

    # The zip function is super handy.  It basically takes a list of tuples and
    # resorts them into a tuple of lists.
    #
    # So e.g. zip((1,2), (3,4), (5,6), (7,8))
    # returns [1,3,5,7], [2,4,6,8]
    #
    # In our case, each run of our simulation is achieved by
    #    calculate_mean_var(make_counts(N, nbins))
    # Each of these is a single mu_i, sigsq_i pair.
    # Then we use list comprehension make a list of M realizations of this.
    #     [... for i in range(M)]
    # Then we use * to split that list into M arguments to the zip function
    #     *[...]
    # Then zip turns these [(mu_0,sigsq_0), (mu_1,sigsq_1), ...]
    #     into [mu_0, mu1, ...], [sigsq_0, sigsq_1, ...]
    # Which is what we want, so return it.
    #
    # The whole thing as a one-liner becomes:

    return zip(*[calculate_mean_var(make_counts(N, nbins)) for i in range(M)])

    # END SOLUTION
```

In [8]:

```
mu, sigsq = run_simulation(100, 1_000_000, 100)

mu_mean, mu_var = calculate_mean_var(mu)
sigsq_mean, sigsq_var = calculate_mean_var(sigsq)

print(f"The mean and variance of mu = {mu_mean:.2f}, {mu_var:.2f}")
print(f"The mean and variance of sigsq = {sigsq_mean:.2f}, {sigsq_var:.2f}")

print(f"I.e. the mean values fall mostly in the range {mu_mean:.2f} +- {np.sqrt
print(f"and the sigsq values fall mostly in the range {sigsq_mean:.2f} +- {np.s

# BEGIN HIDDEN TESTS
nfail = 0

# To TA's: I wasn't as careful here to figure out what tolerances would be safe
# a couple times.  If it passes any time, give them credit.
# (I ran it ~10 times in a row with no failures on my machine, so I think it sh
with log_assert():
    assert np.isclose(mu_mean, 1000, rtol=0.001), mu_mean
with log_assert():
    assert np.isclose(mu_var, 10, rtol=0.5), mu_var
with log_assert():
    assert np.isclose(sigsq_mean, 1000, rtol=0.1), sigsq_mean
with log_assert():
    assert np.isclose(sigsq_var, 20000, rtol=0.5), sigsq_var

print(f"\nTotal of {nfail} test failures")
assert nfail == 0
# END HIDDEN TESTS
```

```
The mean and variance of mu = 1000.37, 7.85
The mean and variance of sigsq = 996.24, 19746.76
I.e. the mean values fall mostly in the range 1000.37 +- 2.80
and the sigsq values fall mostly in the range 996.24 +- 140.52

Total of 0 test failures
```

In [ ]: