APPM 4560 Lab One

Zoe Farmer

February 26, 2024

Note, all code/algorithms are also in Appendix A in full.

1 Part 1: Simulating Random Permutations

In what follows $n \geq 1$ is a given integer. A permutation of the elements in the set $\{1, \ldots, n\}$ is any ordered list of these elements where no element appears repeated.

More generally, there are $n!$ permutations of the elements in the set $\{1, \ldots, n\}$. Choosing one at random is a random permutation, any of which have a $1/n!$ chance to be selected.

Here's a simple way to generate a random permutation of the set.

- Simulate i.i.d. $Uniform(0, 1)$ random variables $U_1, \ldots, U_n$
- Associate with these the random function $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ defined as $f(i) = \#\{j : U_j \leq U_i\}$.
- The random permutation is then $\sigma \Rightarrow (f(1), \ldots, f(n))$.

1.1 Questions

1. Use the above to design an algorithm that produces a random permutation. The input should be some integer $n \geq 1$, and the outpu tis a random permutation of the elements in the aforementioned set. Assume you can simulate any number of i.i.d. uniform random variables in the interval $(0, 1)$.

```
# def permute(n):
#     u_1, ..., u_n = Unif(0, 1)
#     permuted = []
#     for i in 1:size
#         permuted.append(|{u_i <= u_j; j in [0, n]}|)
#     return permuted
def random_permutation(size: int) -> np.ndarray:
    random_unif = np.random.random(size=size)
    permuted_set = np.zeros(size)
    for i, x in enumerate(random_unif):
        permuted_set[i] = len(random_unif[x <= random_unif])
    return permuted_set
```

Let $n = 7$, $\sigma = (6, 7, 2, 5, 1, 4, 3)$, and $m = 6000$.

2. Let X be the number of times that the permutation σ is observed in m runs of your algorithm. What's the distribution of X? What's the expected value of X? Explain.

Any given σ has a $1/n!$ chance of being selected. Therefore the odds of this being selected is $1/n!$. This means that $X \sim Binomial(6000, 1/7!)$. The expected value here is simply $n \cdot p$, or 1.190476, since each trial has $1/n!$ chance of being a "success", and we're running 6000 of them.

3. Let Y be the random variable that counts the number of times you need to run your algorithm until seeing the permutation σ for the first time. What's the distribution of Y? What's the expected value of Y? Explain.

   $Y \sim Geometric(1/n!)$, since each trial is independent and we're interested in how many are required until we see one success. This means that we get a pmf of $(1-p)^{k-1}p$, where p in this case is equal to $1/n!$. This has expected value $1/p$, or $n! = 5040$.

   Let $k = 2000$.

4. Using your algorithm to obtain k independent realizations of X and obtain the histogram associated with these. How does the resulting histogram compare to the theoretical histogram of X. Explain. Display the histogram and the theoretical distribution on the same plot. Make sure to comment on any expected or unexpected behavior.

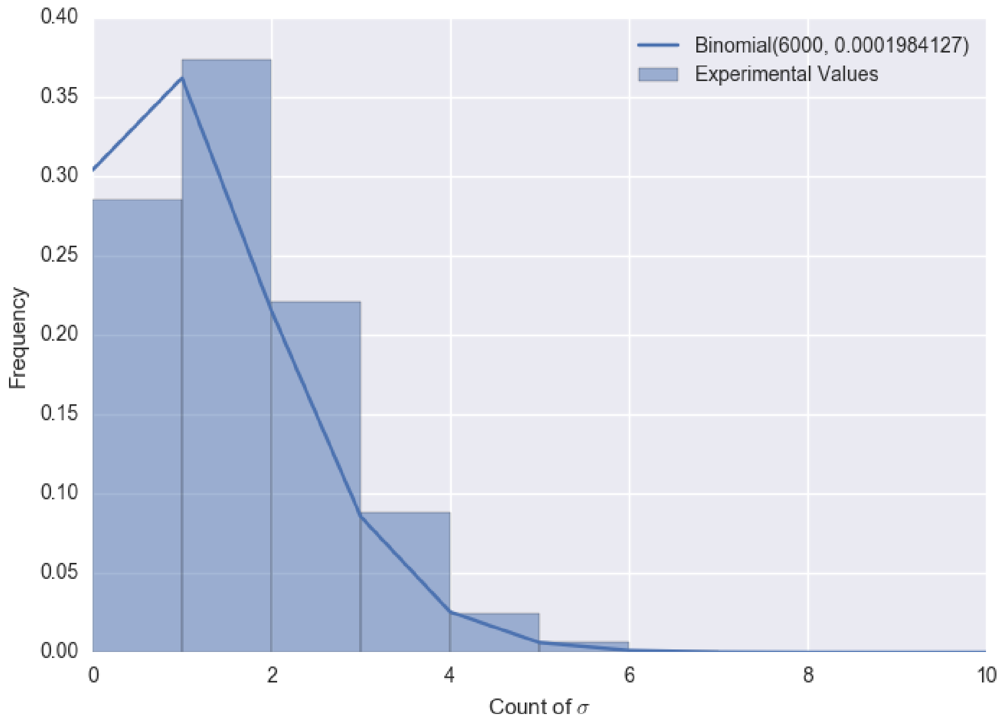

Figure 1: Distribution of X

In Figure 1 we can see that the theoretical distribution of X matches the simulated values almost perfectly. This makes sense, and we would predict this to be the case. It's interesting that the simulated number of zero elements is lower than our distribution predicts, but that just goes to show the issues with simulated distributions.

5. Use your algorithm to obtain k independent realizations of Y and obtain the histogram

associated with these. How does the resulting histogram compare to the theoretical histogram of Y. Explain. Display the histogram and the theoretical distribution on the same plot. Make sure to comment on any expected or unexpected behavior.

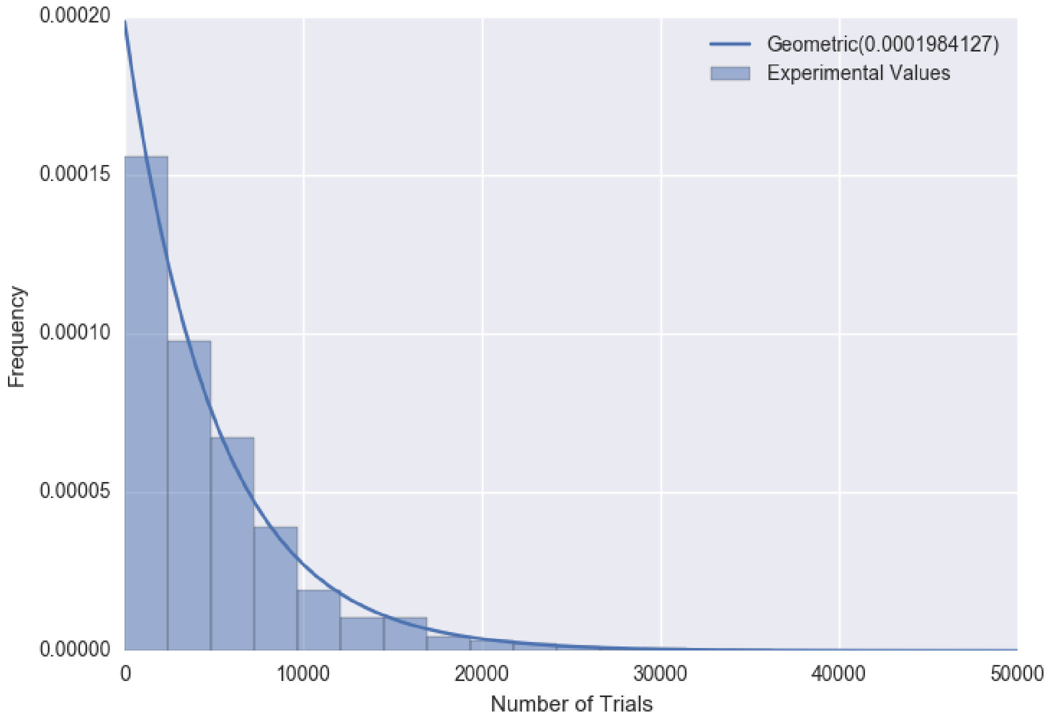

Figure 2: Distribution of Y

In Figure 2 we see that (as before with the distribution of X) the theoretical distribution of Y matches our theory almost perfectly. There are some slight imperfections in the simulated values that lead to some mis-matched values, however on the whole these values are very close.

2 Part 2: Comparing Performance of Retrieving Algorithms

Imagine you have an oracle that always answers the truth to a YES/NO question. Suppose also you have a database with n entries and that your goal is to retrieve a special entry by asking questions to the oracle. Assume the database is a permutation of the elements $\{1, \ldots, n\}$ and that you want to retrieve the position where the element 1 is located.

Here are two methods:

- Iterate and ask about each item
- Binary tree and ask about each half.

Let Q_A denote the total number of questions made to the oracle to determine the position of 1 using method A. Define Q_B similarly.

2.1 Questions

1. Determine $\mathbb{E}(Q_A)$ explicitly. Justify with mathybits.

   $\mathbb{E}(Q_A) = (n/2) + 1$. This can be seen by the following. Assume that for any given random permutation of our set $\mathcal{S}$, the value that we're looking for could be at any position, with uniform distribution. Therefore, the average position is at the halfway mark. This means that on average, if an infinite number of trials occur with our value placed at a uniformly distributed random location location for each trial, the average point will be $(n/2) + 1$. Note, it's not simply $n/2$, as we always have to ask at least one question.

2. Fix $n = 9$ and use your algorithm in the first part to simulate ten-thousand times the random variable Q_A. Determine the sample average of your simulations. Repeat with $n = 21$, $n = 36$, and $n = 69$.

```
data = np.zeros(m)
for i, n in enumerate(n_vals):
    for j in range(m):
        data[j] = np.where(random_permutation(n) == 1)[0][0] + 1
    print(n, data.mean())
```

3. How do the averages compare to your answer in part 1? Summarize your data in a table. Comment on any expected and/or expected behavior.

n	$\mathbb{E}(Q_A)$	Average Iterations
9	4.5	5.00357
21	10.5	10.98759
36	18	18.48431
69	34.5	34.93481

Table 1: Method A Simulated Runtime

   We see that this is almost exactly what we predicted. Interestingly, our values are higher than we'd expect, but this may be a result of the simulation process and errors that occur as a result.

4. What should $\mathbb{E}(Q_B)$ be approximately when n is large?

   $\mathbb{E}(Q_b) = \log_2(n)$. This can be intuitively inferred from realizing that we are constructing a binary tree with depth $\log_2(n)$ when we examine the question space.

5. Repeat 2 above but with Q_B instead of Q_A.

```
def methodB(p: np.ndarray, v: int):
    x0 = 0
    x1 = len(p)
    count = 0
    while True:
        if len(p[x0:x1]) == 1:
            break
        count += 1
        m = x0 + int((x1 - x0) / 2)
        left = p[x0:m]
        right = p[m:x1]
        if v in left:
            x1 = m
        else:
            x0 = m
    return count

# Method B Simulation
data = np.zeros(m)
for i, n in enumerate(n_vals):
    for j in range(m):
        data[j] = methodB(random_permutation(n), 1)
    print(n, data.mean())
```

6. How do the obtained averages compare to your approximation in part 4? Summarize in a table and comment.

n	$\mathbb{E}(Q_B)$	Average Iterations
9	3.1699	3.22071
21	4.3923	4.47676
36	5.1699	5.22242
69	6.1085	6.14644

Table 2: Method A Simulated Runtime

We see that theory here matches our simulated values. This is good. This is exactly as we expected. As the description points out, this is because of the binary tree theory.

A Code

A.1 Part 1

```
#!/usr/bin/env python3.5

import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn
import itertools
import tqdm


# def permute(n):
#     u_1, ..., u_n = Unif(0, 1)
#     permuted = []
#     for i in 1:size
#         permuted.append(|{u_i <= u_j; j in [0, n]}|)
#     return permuted
def random_permutation(size: int) -> np.ndarray:
    random_unif = np.random.random(size=size)
    permuted_set = np.zeros(size)
    for i, x in enumerate(random_unif):
        permuted_set[i] = len(random_unif[x <= random_unif])
    return permuted_set

print(random_permutation(3))
#>> [ 3.  1.  2.]
print(random_permutation(5))
#>> [ 2.  1.  3.  5.  4.]
print(random_permutation(10))
#>> [  4.   8.  10.   1.   6.   2.   9.   7.   3.   5.]
print(random_permutation(20))
#>> [ 17.  11.   7.  14.  16.   2.   8.   4.  13.  18.   1.  15.   6.   3.   9.
#     10.  19.  12.  20.   5.]


n = 7
m = 6000
k = 2000
s = np.array([6, 7, 2, 5, 1, 4, 3])
p = 1 / math.factorial(n)

data = []
for i in tqdm.tqdm(range(k)):
    count = 0
    for j in range(m):
        pp = random_permutation(n)
        if (pp == s).all():
            count += 1
    data.append(count)
hist, bin_edges = np.histogram(data, bins=list(range(11)), density=True)
x = np.arange(0, 11, 1, dtype=int)
binomial = np.zeros(len(x))
for i, k in enumerate(x):
    binomial[i] = ((math.factorial(m) / (math.factorial(k) *
                    math.factorial(m - k))) * p**k * (1 - p)**(m - k))
# plotting
plt.figure()
plt.plot(x, binomial, label='Binomial({}, {:0.08})'.format(m, p))
plt.bar(bin_edges[:-1], hist, width=bin_edges[1] - bin_edges[0],
        alpha=0.5, label='Experimental Values')
plt.ylabel('Frequency')
plt.xlabel(r'Count of $\sigma$')
plt.legend(loc=0)
plt.savefig('distx.png')


n = 7
m = 6000
k = 2000
s = np.array([6, 7, 2, 5, 1, 4, 3])
p = 1 / math.factorial(n)

data = []
for i in tqdm.tqdm(range(k)):
    count = 0
    while True:
        count += 1
        pp = random_permutation(n)
        if (pp == s).all():
            break
    data.append(count)
hist, bin_edges = np.histogram(data, bins=20, density=True)
x = np.arange(0, int(1e5 / 2), 1, dtype=int)
geom = np.zeros(len(x))
for i, k in enumerate(x):
    geom[i] = (1 - p)**(k - 1) * p
# plotting
plt.figure()
plt.plot(x, geom, label='Geometric({:0.08})'.format(p))
plt.bar(bin_edges[:-1], hist, width=bin_edges[1] - bin_edges[0],
        alpha=0.5, label='Experimental Values')
plt.ylabel('Frequency')
plt.xlabel('Number of Trials')
plt.legend(loc=0)
plt.savefig('disty.png')
```

A.2 Part 2

```
#!/usr/bin/env python3.5

import math
import numpy as np
import matplotlib.pyplot as plt
import seaborn
import itertools
import tqdm

def random_permutation(size: int) -> np.ndarray:
    random_unif = np.random.random(size=size)
    permuted_set = np.zeros(size, dtype=int)
    for i, x in enumerate(random_unif):
        permuted_set[i] = len(random_unif[x <= random_unif])
    return permuted_set

n_vals = [9, 21, 36, 69]
m = 100000

data = np.zeros(m)
for i, n in enumerate(n_vals):
    for j in range(m):
        data[j] = np.where(random_permutation(n) == 1)[0][0] + 1
    print(n, data.mean())

print('--------')
def methodB(p: np.ndarray, v: int):
    x0 = 0
    x1 = len(p)
    count = 0
    while True:
        if len(p[x0:x1]) == 1:
            break
        count += 1
        m = x0 + int((x1 - x0) / 2)
        left = p[x0:m]
        right = p[m:x1]
        if v in left:
            x1 = m
        else:
            x0 = m
    return count
```

```
# Method B Simulation
data = np.zeros(m)
for i, n in enumerate(n_vals):
    for j in range(m):
        data[j] = methodB(random_permutation(n), 1)
    print(n, data.mean())
```