IEOR 4404 Simulation **Fall 2024**

Homework 2

Instructor: Henry Lam

Problem 1 Let $X \sim Bin(n,p)$ be a binomial random variable with parameters n and p. Recall that the p.m.f of X is given by

$$p(k) = \mathbb{P}(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, 2, \cdots, n.$$

(a) Verify that

$$p(k+1) = \frac{n-k}{k+1} \cdot \frac{p}{1-p} \cdot p(k), \quad k = 0, 1, \cdots, n-1.$$

(b) Write an algorithm (or "pseudo code") for the inverse transform method for generating X that utilizes the relation in part (a), in a way that you do not have to compute all the p.m.f. values, and also you only have to compute the c.d.f. values on the fly if needed.

(c) Implement your answer in part (b) on a computer to generate 100 copies of $Bin(10, 2/3)$ and plot their distribution.

Problem 2 In this problem we will generate a negative binomial random variable with parameters r, p in three different ways. This random variable, called $NB(r,p)$, is the number of independent Bernoulli trials needed to get r successes, where each Bernoulli trial has success probability p. It has a p.m.f. given by

$$p(k) = \binom{k-1}{r-1} p^r (1-p)^{k-r}, k = r, r+1, \ldots$$

(a) Verify the relation

$$p(k+1) = \frac{k(1-p)}{k+1-r} p(k).$$

(b) Use the relation in part (b) to give an algorithm for generating $NB(r,p)$.

(c) Write down the relationship between $NB(r,p)$ and $Geom(p)$. Use it to obtain an algorithm to generate $NB(r,p)$.

(d) Using the interpretation that $NB(r,p)$ counts the number of independent $Ber(p)$ trials required to accumulate r successes, obtain yet another approach for generating $NB(r,p)$.

(e) Implement your algorithms in parts (a), (b) and (c) on a computer for generating $NB(2, 1/3)$. For each algorithm, generate 100 copies of X and plot their distribution. Are the plots similar?

Problem 3 Write an algorithm to generate the random variable X where

$$\mathbb{P}(X = j) = \left(\frac{1}{2}\right)^{j+1} + \left(\frac{1}{2}\right) \frac{2^{j-1}}{3^j}, j = 1, 2, \ldots$$

Implement your algorithm on a computer to generate 100 copies of X, and plot their distribution.

Problem 4 A fair die is to be continually rolled until all possible outcomes $1, 2, \ldots, 6$ have occurred at least once, and we are interested in the total number of rolls in this experiment. Implement an algorithm on a computer to generate 100 copies of the total number of rolls, and plot the distributions of these 100 copies.

Alexandre
Duhamel
afd2153
MSOR

HW2 : Simulation

Problem 1:

a) We know that $p(k) = \binom{m}{k} p^k (1-p)^{m-k}$ for $k: 0, 1, \dots, m$

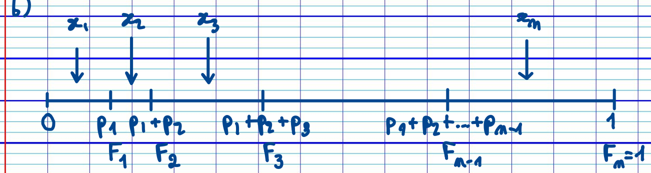
Thus $\boxed{p(k+1)} = \binom{m}{k+1} p^{k+1} (1-p)^{m-k-1}$ for $k: 0, 1, \dots, m-1$

$$= \frac{m!}{(k+1)!(m-k-1)!} p^k \cdot p \frac{(1-p)^{m-k}}{1-p}$$

$$= \frac{m!}{k!(m-k)!} \frac{m-k}{k+1} p^k (1-p)^{m-k} \frac{p}{1-p}$$

$$= \boxed{\frac{m-k}{k+1} \frac{p}{1-p} p(k)}$$

b)



Input : m, p

Initialize : $\begin{cases} \text{Set } F = p(0) \\ \text{Generate } U \sim \text{Unif}(0,1) \\ k = 0 \end{cases}$

While $F < U$: $\begin{cases} \text{update } p(k+1) = \frac{m-k}{k+1} \frac{p}{1-p} p(k) \\ \text{update } F \mathrel{+}= p(k+1) \\ k{+}{+} \end{cases}$

Return k

c)

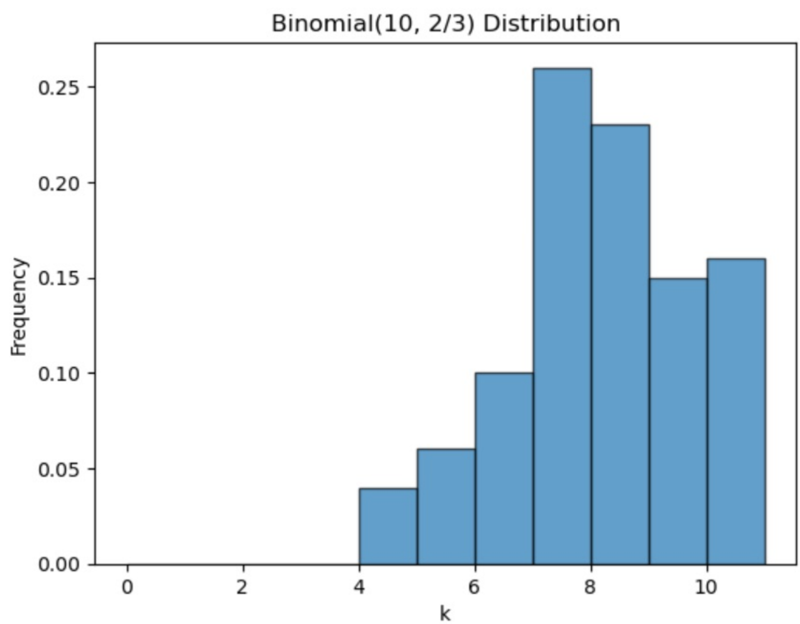
```python
def binomial_inverse_transform(n, p, num_samples=100):
    samples = []

    # Loop for samples
    for _ in range(num_samples):
        # Initialize
        F = 0
        k = 0
        prob_k = (1 - p)**n
        U = np.random.uniform()

        # While for probability
        while F < U:
            F += prob_k
            prob_k *= (n - k) / (k + 1) * p / (1 - p)
            k += 1

        samples.append(k)

    return samples

# Parameters
n = 10
p = 2/3

# Generate 100 samples
samples = binomial_inverse_transform(n, p, 100)

# Plot the distribution
plt.hist(samples, bins=range(n+2), density=True, edgecolor='k', alpha=0.7)
plt.title("Binomial(10, 2/3) Distribution")
plt.xlabel("k")
plt.ylabel("Frequency")
plt.show()
```

do the experiment multiple times

previous pseudo-code formalized

store the results

basically counts the number of appearances of all X's and normalizes it to create a density



Problem 2:

a) We know that $p(k) = \binom{k-1}{r-1} p^r (1-p)^{k-r}$ for $k = r, r+1, \dots$

Thus $\boxed{p(k+1)} = \binom{k}{r-1} p^r (1-p)^{k+1-r}$

$$= \frac{k!}{(r-1)!\,(k-r+1)!} p^r (1-p)^{k-r} (1-p)$$

$$= \binom{k-1}{r-1} \frac{k}{k-r+1} p^r (1-p)^{k-r} (1-p)$$

$$= \boxed{\frac{k}{k-r+1} (1-p)\, p(k)}$$

b)

Input : r, p

Initialize : $\begin{cases} \text{Set } p(k) = p(r) = p^r \text{ and } F = p(k) \\ \text{Generate } U \sim Unif(0,1) \\ k = r \end{cases}$

While $F < U$: $\begin{cases} \text{update } p(k+1) = \frac{k(1-p)}{k+1-r} p(k) \\ \text{update } F \mathrel{+}= p(k+1) \\ k{+}{+} \end{cases}$

Return k

c) Negative binomial experiment counts the number of trials required to achieve a fixed number of successes. Thus, it can be tought of as the sum of r independant $Geo(p)$ (where each $Geo(p)$ counts the number of trials until a success w.p. p).

The idea is : $\begin{cases} \text{Generate } r \text{ independent } Geo(p) \text{ RV} \\ \text{Sum these RV to get } NegBin(r,p) \end{cases}$

Input : r, p

Initialize : Set $X = 0$

For $i = 1 \dots r$: Generate $G_i \sim Geo(p)$:
$X \mathrel{+}= G_i$

since we want r successes

Return X

$G_i \sim Geo(p)$:

- $I, G = 0, 1$
- while $I = 0$:
  - $U \sim Unif(0,1)$
  - if $U \leq p$:
    - $I = 1$
  - else $G \mathrel{+}= 1$.
- return G

instead of using $I = 0/1$ we could simply put a loop break :)

d) Back on the idea that NegBin counts the number of trials needed to achieve r successes:

Input : r, p

Initialize : win = 0 , try = 0

While win < r : $\begin{cases} \mathcal{U} \sim \text{Unif}(0,1) \\ \text{if } \mathcal{U} \leq p : \\ \quad \text{win} += 1 \\ \text{try} += 1 \end{cases}$

Return try

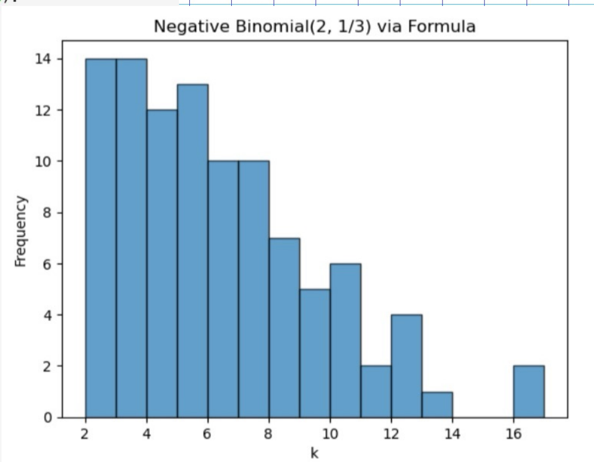
e) a.

Problem 2, question e

```
# Parameters
r = 2
p = 1/3
```

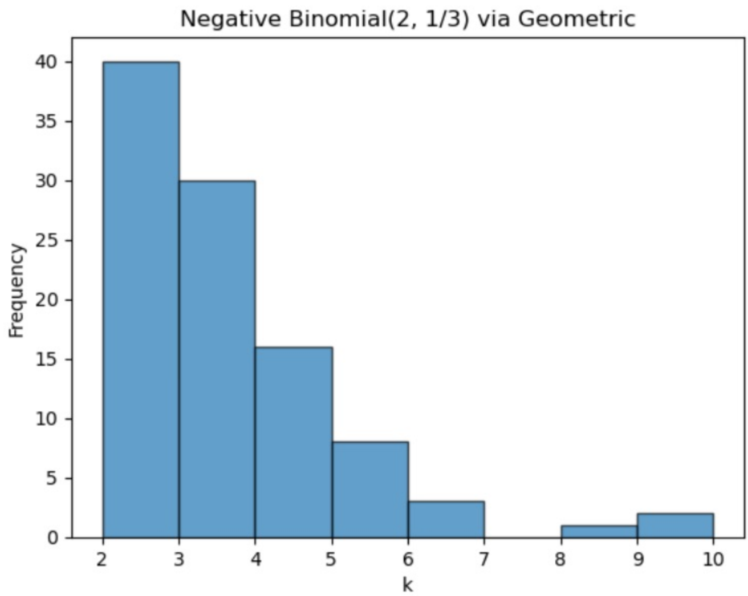
Method 1 --> formula

```
import numpy as np
import matplotlib.pyplot as plt

def negative_binomial_recursive(r, p, num_samples=100):
    samples = []

    # Loop for samples
    for _ in range(num_samples):
        F = 0
        k = r
        prob_k = p**r  # p(k = r)

        U = np.random.uniform()

        # While for probability
        while F < U:
            F += prob_k
            if F >= U:
                break
            prob_k *= k * (1 - p) / (k + 1 - r)
            k += 1

        samples.append(k)

    return samples

# Generate 100 samples
samples_recursive = negative_binomial_recursive(r, p, 100)

# Plot the distribution
plt.hist(samples_recursive, bins=range(min(samples_recursive), max(sample
plt.title("Negative Binomial(2, 1/3) via Formula")
plt.xlabel("k")
plt.ylabel("Frequency")
plt.show()
```



Method 2 --> geometric

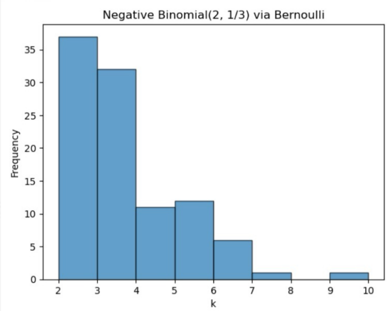
```python
def gen_geo(p):
    win, count = 0, 1
    while win == 0:
        U = np.random.uniform()
        if U <= p:
            win = 1
        else :
            count += 1
    return count

def negative_binomial_geom(r, p, num_samples=100):
    samples = []

    # Loop for samples
    for _ in range(num_samples):
        total = 0

        # Genearating Geo(p)
        for _ in range(r):
            geom = gen_geo(p)
            total += geom
        samples.append(total)

    return samples

# Generate 100 samples
samples_geom = negative_binomial_geom(r, p, 100)

# Plot the distribution
plt.hist(samples_geom, bins=range(min(samples_geom), max(samples_geom)+2),
plt.title("Negative Binomial(2, 1/3) via Geometric")
plt.xlabel("k")
plt.ylabel("Frequency")
plt.show()
```



Method 3 --> bernoulli

c.

```
def negative_binomial_bernoulli(r, p, num_samples=100):
    samples = []

    # Loop for samples
    for _ in range(num_samples):
        successes = 0
        trials = 0

        # While for probability
        while successes < r:
            if np.random.uniform() <= p:
                successes += 1
            trials += 1
        samples.append(trials)

    return samples

# Generate 100 samples
samples_bernoulli = negative_binomial_bernoulli(r, p, 100)

# Plot the distribution
plt.hist(samples_bernoulli, bins=range(min(samples_bernoulli), max(sampl
plt.title("Negative Binomial(2, 1/3) via Bernoulli")
plt.xlabel("k")
plt.ylabel("Frequency")
plt.show()
```



– Of course, all three graphs differ a little due to the randomness of U but they still share similarities such as a concentration of the density in lower values of k, then decreasing as k increases. They all have a peak value before $k=5$ showing that their underlying algorithms model a situation quite close (and decreasing)
(We could increase n even further to convince ourselves).

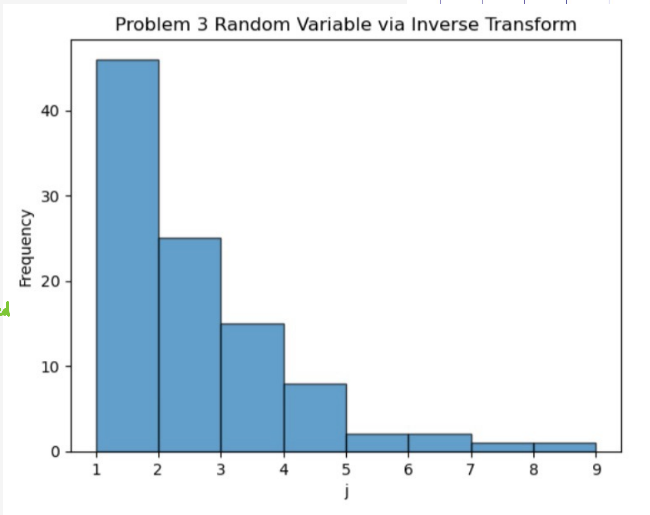
Problem 3:

Remember that $CDF = \mathbb{P}(X \leq i) = \sum_{k=1}^{i} \mathbb{P}(X = k)$

So we simply have to sum the probabilities via the formula on the fly.

```python
def compute_p(j):
    prob = (1/2)**(j+1) + (1/2)*(2**(j - 1)) / (3**j)
    return prob

def inverse_transform(num_samples=100):
    samples = []

    # Loop for samples
    for _ in range(num_samples):
        U = np.random.uniform()
        F = 0
        j = 1

        # While for probability
        while F < U:
            prob = compute_p(j)
            F += prob
            if F < U:
                j += 1
        samples.append(j)

    return samples

# Parameters
num_samples = 100

# Generate samples
samples = inverse_transform(num_samples)

# Plot the distribution
plt.hist(samples, bins=range(min(samples), max(samples)+2), edgecolor='k',
plt.title("Problem 3 Random Variable via Inverse Transform")
plt.xlabel("j")
plt.ylabel("Frequency")
plt.show()
```

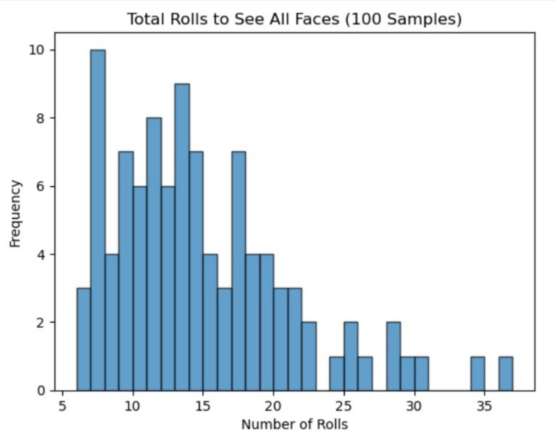
since we technically computed the next F_k



Problem 4:

- My initial idea was to use a dictionary to count if a face appeared:

```
def inverse_transform(num_samples):
    samples = []

    # Loop for samples
    for _ in range(num_samples):
        dice = {i: 0 for i in range(1, 7)}
        k = 0

        # While for probability
        while 0 in dice.values():
            U = np.random.uniform()
            throw = int(np.floor(6 * U)) + 1

            if dice[throw] == 0:
                dice[throw] = 1

            k += 1

        samples.append(k)

    return samples

# Parameters
num_samples = 100

# Generate samples
samples = inverse_transform(num_samples)

# Plot the distribution of the total number of rolls
plt.hist(samples, bins=range(min(samples), max(samples)+2), edgecolor='k',
plt.title("Total Rolls to See All Faces (100 Samples)")
plt.xlabel("Number of Rolls")
plt.ylabel("Frequency")
plt.show()
```



I then realized that I could use a set to know if a face appeared, utilizing its structure to remove duplicates:

```python
import numpy as np
import matplotlib.pyplot as plt

def inverse_transform(num_samples):
    samples = []

    # Loop for samples
    for _ in range(num_samples):
        faces = set()
        k = 0

        # While for probability
        while len(faces) < 6:  # Continue rolling until all six faces have appeared
            U = np.random.uniform()
            throw = int(np.floor(6 * U)) + 1
            faces.add(throw)
            k += 1

        samples.append(k)

    return samples

# Parameters
num_samples = 100

# Generate 100 samples
samples = inverse_transform(num_samples)

# Plot the distribution of the total number of rolls
plt.hist(samples, bins=range(min(samples), max(samples)+2), edgecolor='k', alpha=0.7)
plt.title("Total Rolls to See All Faces (100 Samples)")
plt.xlabel("Number of Rolls")
plt.ylabel("Frequency")
plt.show()
```

simply add throw to the set