PROBABILITY AND RANDOM PROCESSES

EE325 PROGRAMMING ASSIGNMENT - 2

Aminesh Gogia - 23B1210
Rushabh Gayakwad - 23B2464
Jaswin Reddy M - 23B1289

Question - 1

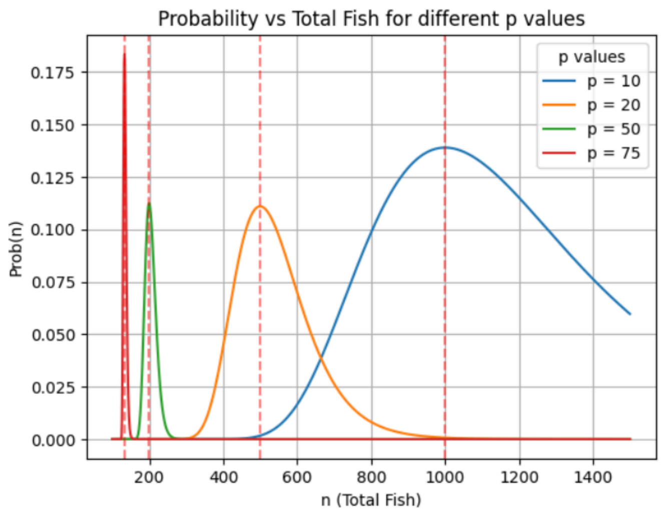
The probability that we get back exactly p of our m marked fish is given by:

$$\frac{\binom{m}{p} \cdot \binom{n-m}{m-p}}{\binom{n}{m}}$$

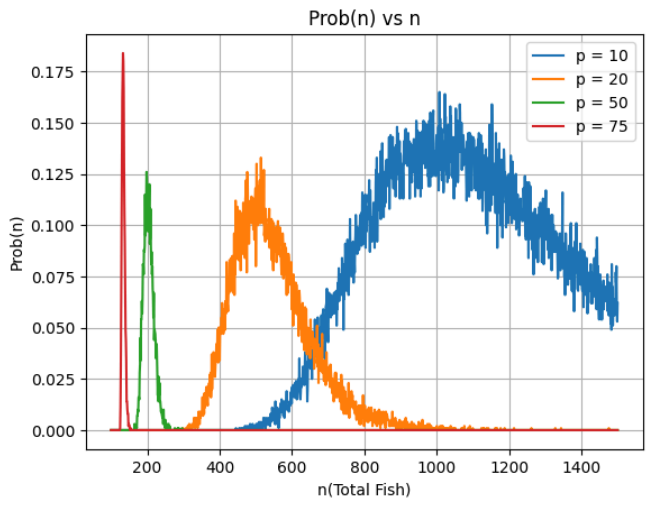
where:

- $\binom{m}{p}$ is the number of ways to choose p marked fish from m,
- $\binom{n-m}{m-p}$ is the number of ways to choose $m-p$ unmarked fish from the remaining $n-m$ fish.
- $\binom{n}{m}$ is the total number of ways to choose m fish from the n fish.

For each *m*, we plotted graphs of this theoretical probability vs *n*, and also by simulating selection of *m* fish for different values of *n*, from 100 to 1500. We obtained peaks at similar values in both cases.



(Theoretical)



(Simulated)

'Favorable' values: n1 = 999, n2 = 499, n3 = 199, n4 = 133.

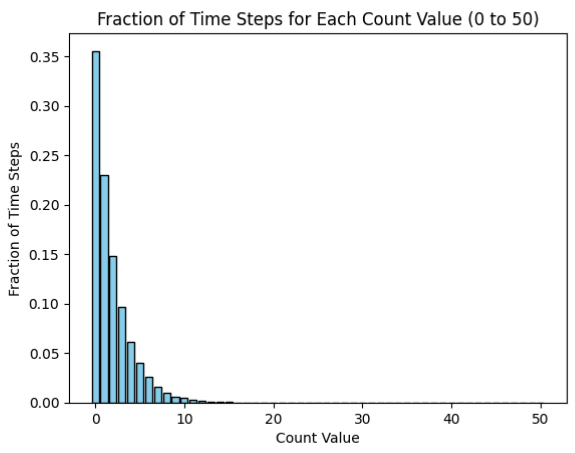
Why are the peaks favorable?

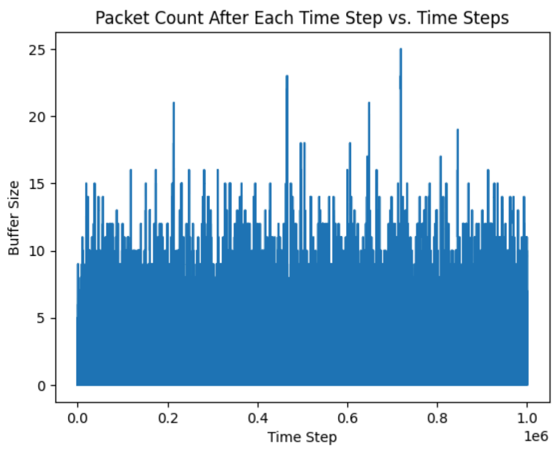
For a given m, the conditional probability that $n = n_0$ given that exactly p marked fish were obtained is (using Bayes' Theorem):

$$P(n = n_0 \mid \text{exactly } p \text{ marked fish obtained}) = \frac{\binom{m}{p} \cdot \binom{n_0 - m}{m - p}}{\sum_k \left(\binom{m}{p} \cdot \binom{k-m}{m-p} \right)}$$

Now, for a given m and p, the denominator is the same for different values of n0. So, the conditional probability for *n = no* given that exactly *p* fish are recovered is maximum at the peaks of the graphs plotted, thus they are the 'favorable', most probable values.

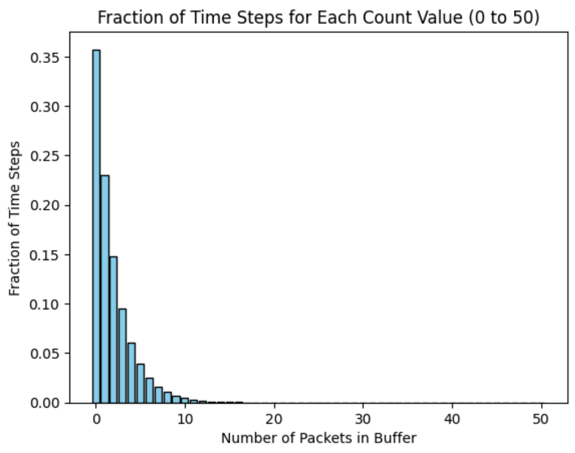
Question - 2

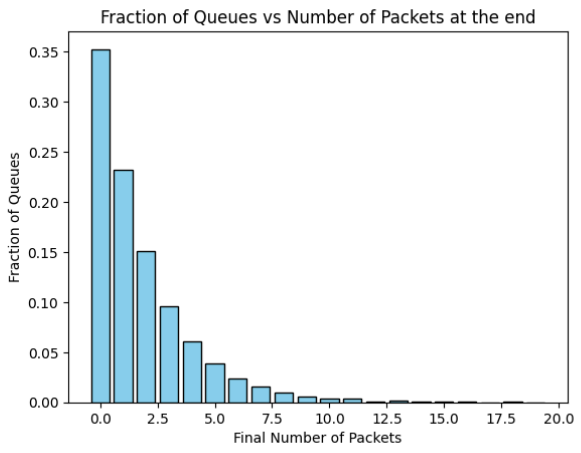




Plotting the fraction of time steps for each buffer size, we see a declining trend, which is intuitive, since it is more likely for packets already in the buffer to leave than for new packets to be added. The average number of packets in the buffer over all time steps is a low 1.78, as the number of packets in the buffer oscillates with a low amplitude, rising up slightly and coming back to zero vigorously.

Question - 3





Averaging over multiple queues, we again get a similar plot of fractions of time steps vs number of packets in the buffer. At the end of 100,000 time steps, most queues have a low number of packets in the buffer, and the number of queues decreases with the number of packets at the end.

Question - 4

1. Probability Mass Function (PMF) of a Binomial Random Variable:

$$P(X=k)=\binom{n}{k}p^k(1-p)^{n-k}$$

Where:

- $\binom{n}{k}$ is the binomial coefficient, calculated as $\frac{n!}{k!(n-k)!}$.
- p (= 0.5 + c) is the probability of a member making the correct decision.
- **k** is the number of correct decisions.
- **n** is the total number of jurors.

2. Probability that a Majority of Jurors Make the Correct Decision:

The probability that the majority of the jurors are correct is given by:

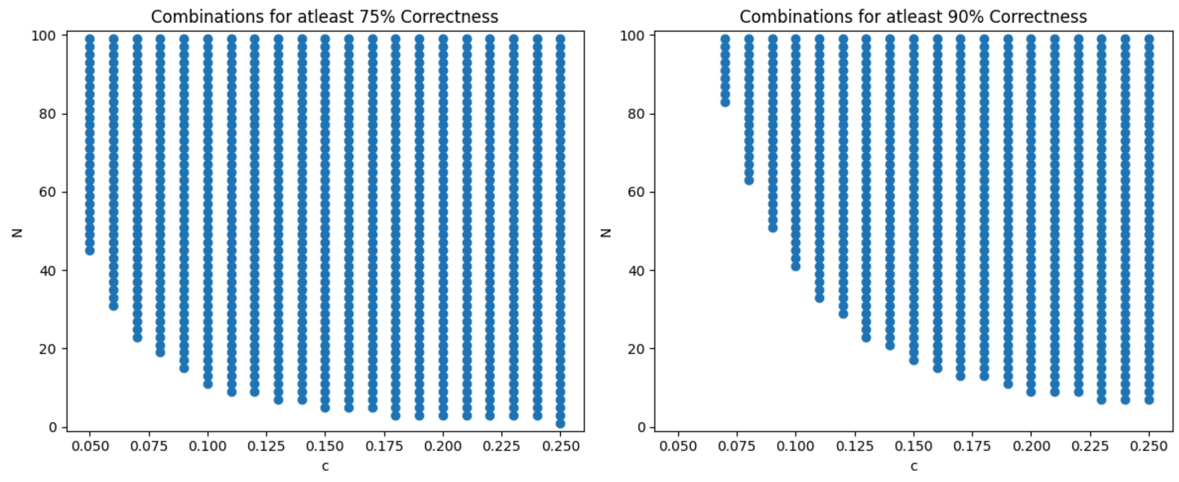
$$P\left(X > \frac{N}{2}\right) = \sum_{k=\left\lceil \frac{N}{2} \right\rceil}^{N} \binom{N}{k} p^k (1-p)^{N-k}$$

Where:

- $\left\lceil \frac{N}{2} \right\rceil$ is the smallest integer greater than or equal to $\frac{N}{2}$.
- The summation accumulates the probabilities of all outcomes where the correct decisions exceed half of the jurors.

Discussion on Findings

The plots generated from the code show possible combinations of c (talent of juror) and N (jury size) for the two thresholds:



Suggested Cost Function

$$Cost = mN^2 + ne^{kc}$$

where:

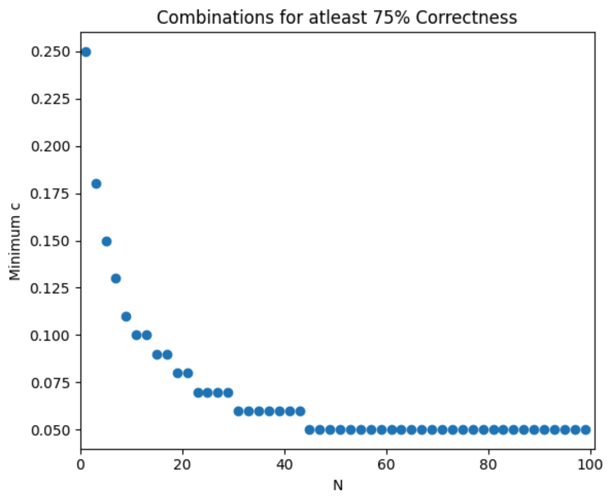
- **m** represents the logistic cost per juror due to size (quadratic growth to show increased complexity with larger N)
- **n** is a parameter that scales the overall cost of recruiting jurors
- **k** represents how quickly the cost of jurors increases as c increases

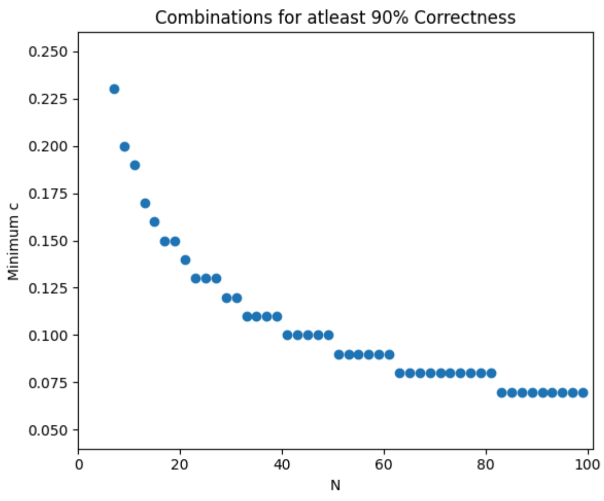
Values of m, n and k can be selected by reviewing datasets on jury recruiting and management.

Selection of Parameters

1. Assuming jury logistics are moderately expensive, so m=0.1.
2. Juror recruitment costs are manageable, so n=10.
3. Intelligent jurors are expensive and rare, so k=20.

Optimal Values of c and N (with example parameters)





- For at least 75% correctness: c = 0.09 and N = 15.0 minimizes the cost to 83.00.
- For at least 90% correctness: c = 0.13 and N = 23.0 minimizes the cost to 187.54.

For 75% Correctness Requirement:

- Smaller values of c (closer to 0.05) require larger jury sizes to reach the required correctness probability.

- As c increases, a lesser number of jurors are needed. For example, with c = 0.05, a larger N (45 or more jurors) is necessary. However, with c = 0.25, even a smaller jury size (1-3 jurors) is sufficient.

For 90% Correctness Requirement:

- Larger jury sizes are necessary even when c is moderately high. For example, with c=0.10, approximately 45 jurors are needed, while with c=0.25, about 9 jurors are enough.
- **Higher N values** make it possible to achieve the desired correctness even with lower values of c, showing that increasing the jury size makes up for less talented jurors.
- **Higher c values** are effective at reducing the required jury size, making the decision process more efficient, but jurors with higher c are harder to find and might be costly.

compiled-assignment

September 13, 2024

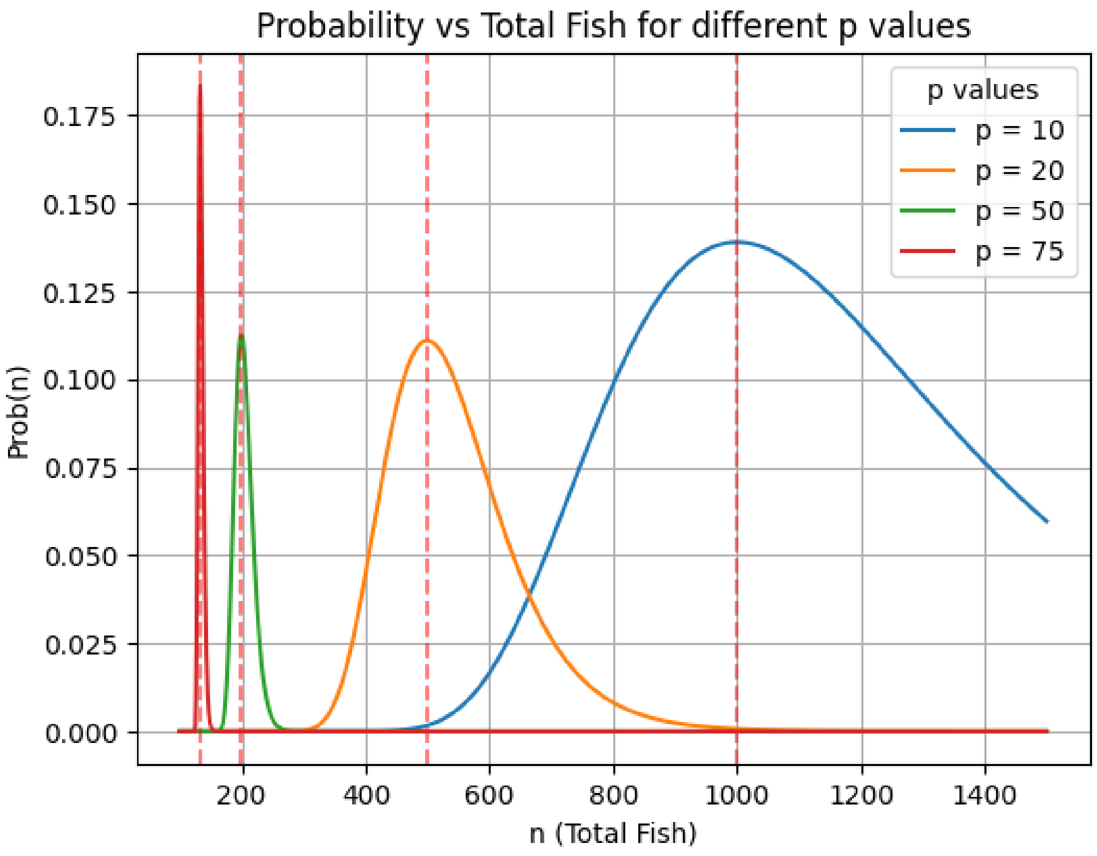
1 EE325 Programming Assignment - 2

1.1 Question 1:

1.1.1 Plotting the mathematical formula:

[21]:
```
import math
import matplotlib.pyplot as plt

m = 100
p = [10, 20, 50, 75]
combinations = math.comb
x = [_ for _ in range(100, 1500)]

for j in p:
    max_pr = 0
    prob = []
    for i in range(100, 1500):
        tot = combinations(i, m) # iCm
        tag_fish = combinations(m, j) # mCj
        catch_fish = combinations(i - m, m - j) # (i -m)C(m - j)

        pr = (tag_fish * catch_fish) / tot # the prob
        prob.append(pr)

        if pr > max_pr:
            max_pr = pr
            max_index = i

    print(f"Favorable 'n{p.index(j) + 1}' is: {max_index}")
    plt.plot(x, prob, label=f'p = {j}')
    plt.axvline(max_index, c='r', linestyle='--', alpha=0.5)

plt.legend(title='p values')
plt.xlabel('n (Total Fish)')
plt.ylabel('Prob(n)')
plt.grid(True)
plt.title('Probability vs Total Fish for different p values')
plt.show()
```

```
Favorable 'n1' is: 999
Favorable 'n2' is: 499
Favorable 'n3' is: 199
Favorable 'n4' is: 133
```



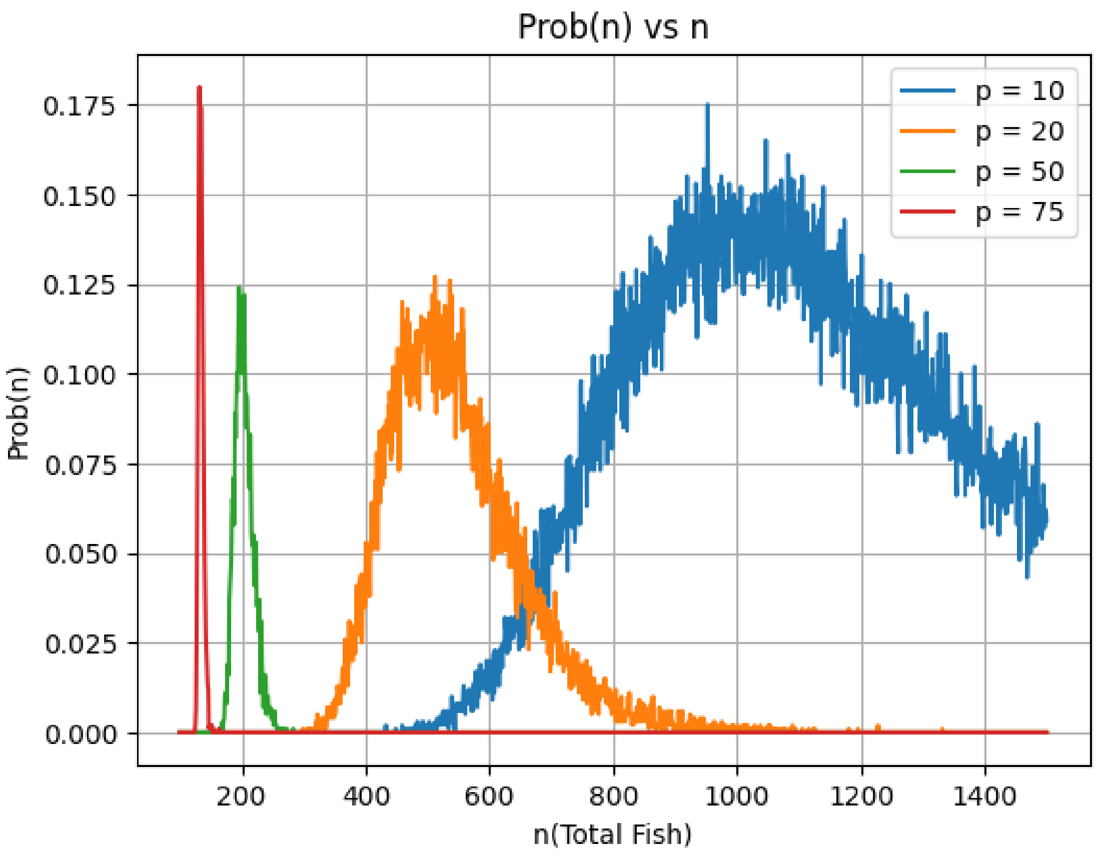
1.1.2 Simulating the Experiment

[22]:
```
import random

m = 100
N = [_ for _ in range(100,1500)]          # Test Range of values for 'n', Total␣
 ↪Fish in the lake
P = [10,20,50,75]                         # Marked Fishes, 'm'
iterations = 1000                         # Iterations ~ Experiments needed to␣
 ↪conclude estimate 'n'

observations = [[] for _ in range(4)]    # Fishes that are marked with suitable P
```

```python
for n in N:
  marked = random.sample(range(n), k=m)
  result = [0 for _ in range(4)]         # Checking how many Fishes are marked␣
 ↪with total count in P
  for iter in range(iterations):
    print(f'N={n} || Iterations = {iter}',end='\r')
    sample = random.sample(range(n), k=m)
    count = 0
    for s in sample:
      if s in marked:
        count += 1
    for p in P:
      if count == p:
        result[P.index(p)] += 1

  for p in P:
    observations[P.index(p)].append(result[P.index(p)]/iterations)
```

```
N=1499 || Iterations = 999
```

[23]:
```python
import matplotlib.pyplot as plt
for p in P:
  plt.plot(N, observations[P.index(p)], label=f'p = {p}')
plt.title('Prob(n) vs n')
plt.xlabel('n(Total Fish)')
plt.ylabel('Prob(n)')
plt.grid(True)
plt.legend()
plt.show()
```



Peaks are close to the theoretically computed peak n values. (approximately n1 = 1000, n2 = 500, n3 = 200, n4 = 100)

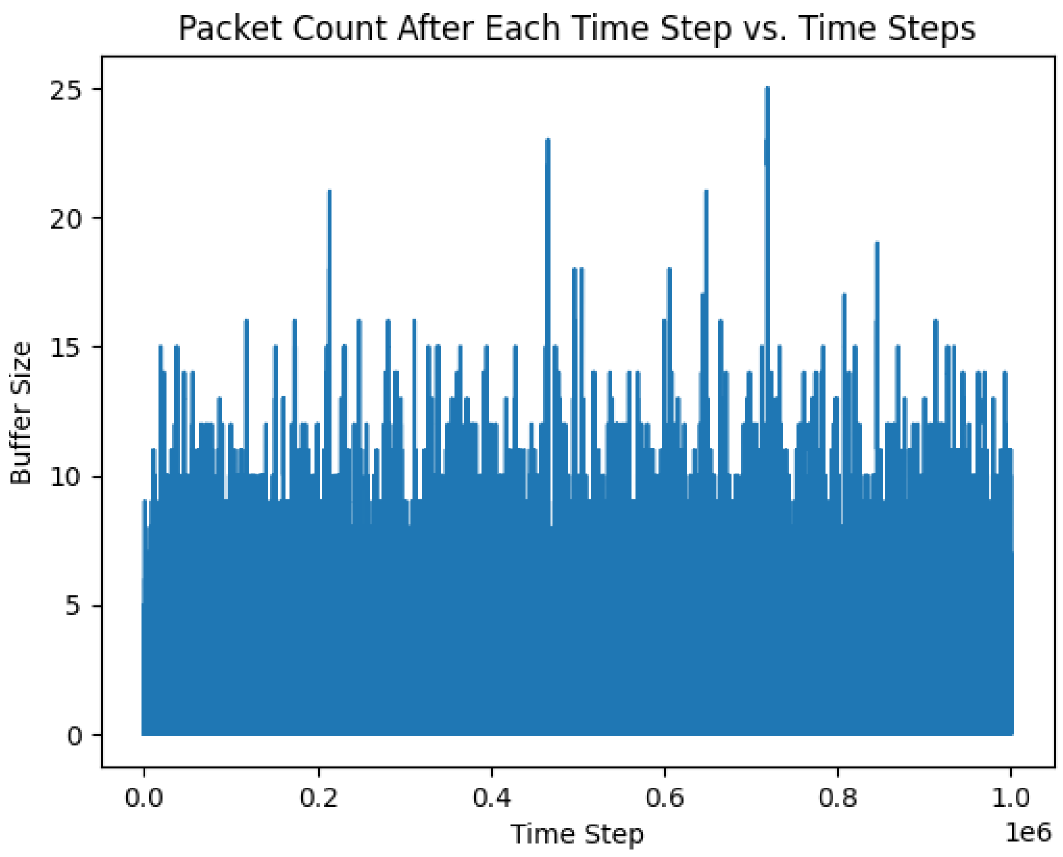
1.2 Question 2:

1.2.1 Plotting fraction of time for which buffer has a certain number of packets

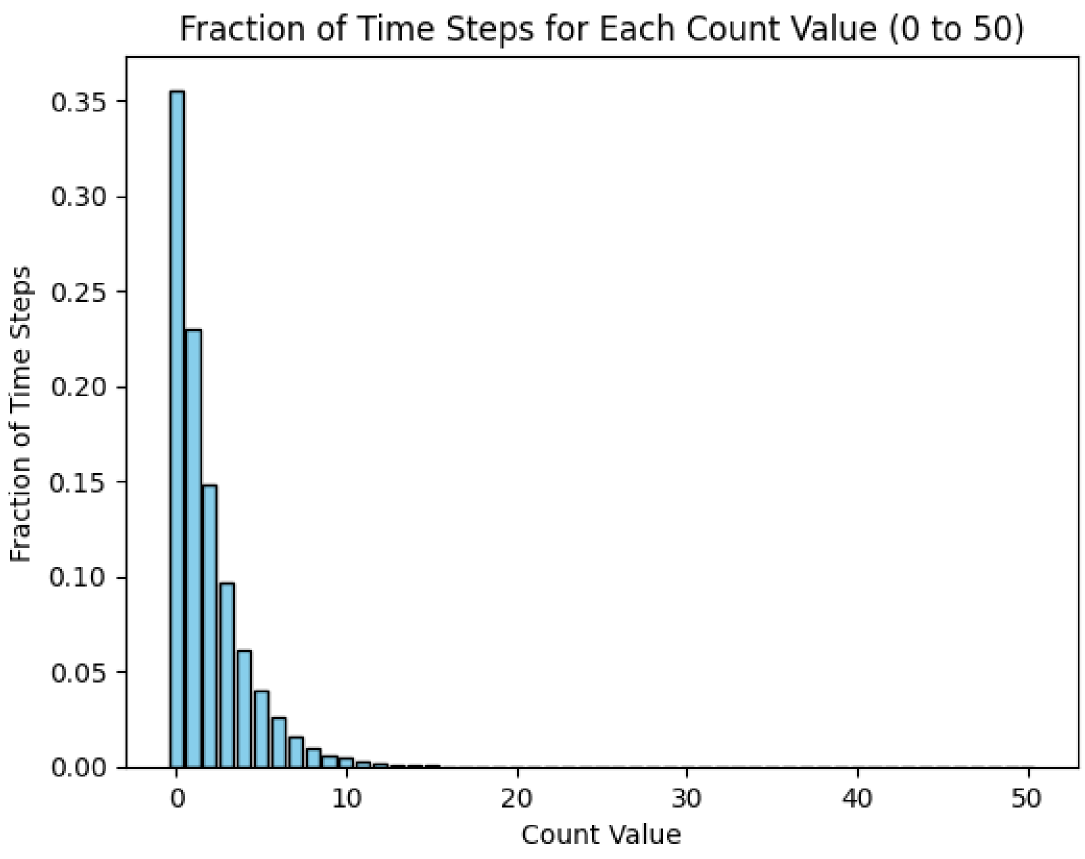
[24]:
```
import numpy as np
np.random.seed(10) ## For consistency of results

lamb = 0.3 ## Probability that a packet arrives at a given time step
mu = 0.4 ## Probability that a packet leaves at a given time step
num_of_time_steps = 1000000

p_dec = (1 - lamb) * mu ## Probability that buffer size decreases after one␣
  ↪time step
p_inc = lamb * (1 - mu) ## Probability that buffer size increases after one␣
  ↪time step
p_same = 1 - p_dec - p_inc ## Probability that buffer size remains the same␣
  ↪after one time step
```

```
## Simulating the change in buffer size at each time step
each_step = np.random.choice([-1, 0, 1], size = num_of_time_steps,  p = [p_dec,␣
 ↪p_same, p_inc])

count_after_each_step = np.zeros(num_of_time_steps + 1, dtype= int)

## Storing the buffer size after each step
for i in range(1, num_of_time_steps + 1):
    count_after_each_step[i] = count_after_each_step[i - 1] + each_step[i - 1]
    if count_after_each_step[i] < 0 : count_after_each_step[i] = 0
```

[25]:
```
plt.plot(range(num_of_time_steps + 1), count_after_each_step)
plt.title('Packet Count After Each Time Step vs. Time Steps')
plt.xlabel('Time Step')
plt.ylabel('Buffer Size')
plt.show()
```



[26]:
```
counts_of_interest = np.bincount(count_after_each_step)
counts_in_range = np.pad(counts_of_interest, (0, max(0, 51 -␣
 ↪len(counts_of_interest))), 'constant')[:51]/num_of_time_steps

x_values = np.arange(51)

plt.bar(x_values, counts_in_range, color='skyblue', edgecolor='black')
plt.title('Fraction of Time Steps for Each Count Value (0 to 50)')
plt.xlabel('Count Value')
plt.ylabel('Fraction of Time Steps')
plt.show()
```



[27]:
```
np.mean(count_after_each_step) ## Average number of packets in the buffer over␣
 ↪the entire time interval
```

[27]:
```
1.7872792127207873
```

1.3 Question 3:

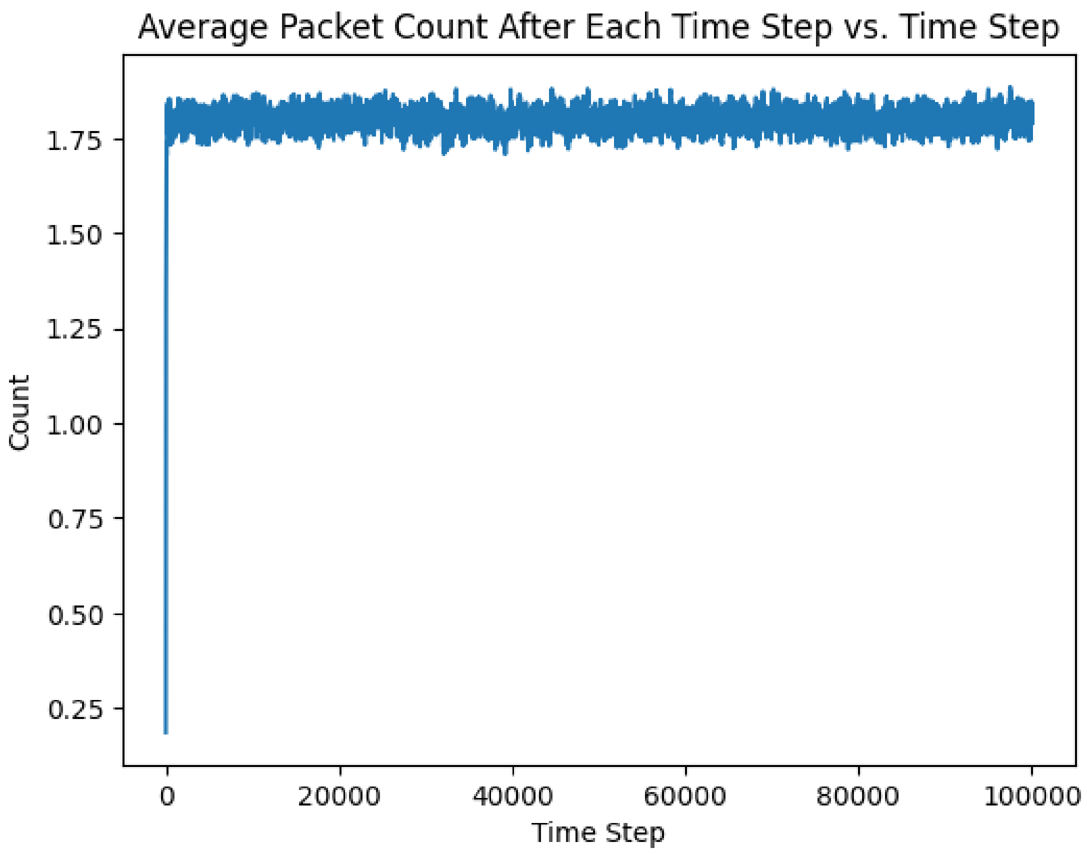
1.3.1 Simulating multiple queues in parallel, and checking time fraction of a certain buffer size

[28]:
```
num_of_time_steps = 100000
count_in_queues = np.zeros(10000, dtype= int)
num_of_parallel_queues = 10000
counts_after_each_step = []
d = {i: 0 for i in range(0, num_of_time_steps )} ## Stores the number of times␣
 ↪a particular buffer size is observed, across all queues

for i in range(num_of_time_steps):
    ## Simulating the queues in parallel
    count_in_queues += np.random.choice([1, 0], size= num_of_parallel_queues,␣
 ↪p=[lamb, 1 - lamb]) - np.random.choice([1, 0], size =␣
 ↪num_of_parallel_queues, p=[mu, 1 - mu])
    count_in_queues = np.maximum(count_in_queues, 0)
    counts_after_each_step.append(np.array(count_in_queues))
    unique_counts, counts = np.unique(count_in_queues, return_counts=True)

    for unique_val, count in zip(unique_counts, counts):
        d[unique_val] += count
```

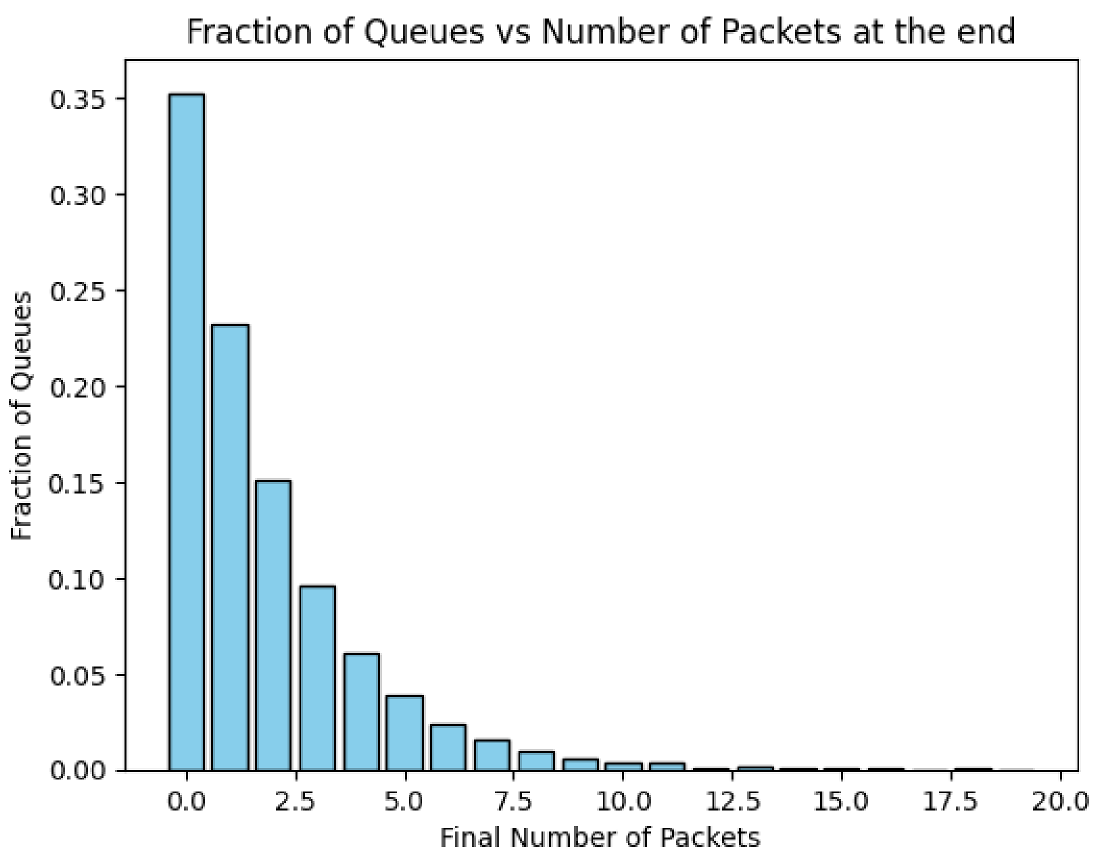
[29]:
```
## Average queue size at each time step across queues
mean_count_post_each_step = [np.mean(x) for x in counts_after_each_step]

## Average at the end, after all time steps
mean_count_post_each_step[-1]
```

[29]:
```
1.8042
```

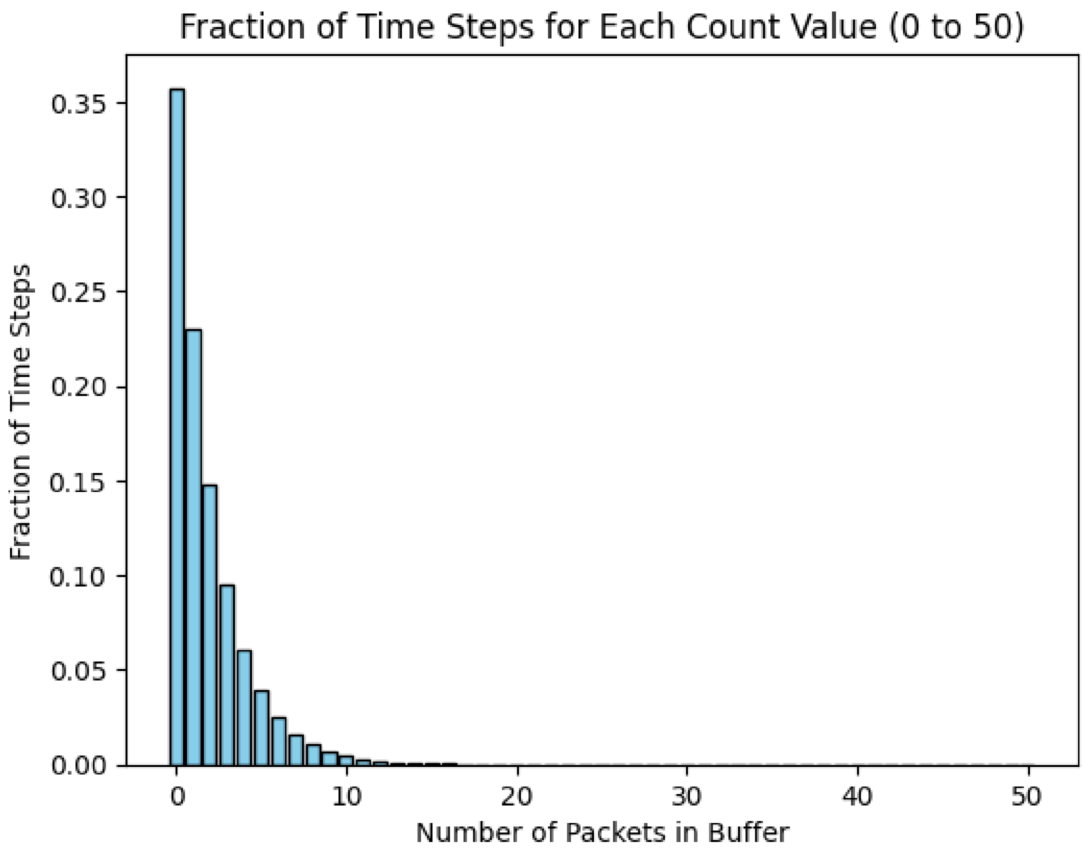
[30]:
```
plt.plot(range(1, num_of_time_steps + 1), mean_count_post_each_step)
plt.title('Average Packet Count After Each Time Step vs. Time Step')
plt.xlabel('Time Step')
plt.ylabel('Count')

plt.show()
```



[31]:
```
l = [int(c) for c in count_in_queues]
data = {i: l.count(i)/num_of_parallel_queues for i in set(l)}

keys = list(data.keys())
values = list(data.values())

plt.figure()
plt.bar(keys, values, color='skyblue', edgecolor='black')
plt.title('Fraction of Queues vs Number of Packets at the end')
plt.xlabel('Final Number of Packets')
plt.ylabel('Fraction of Queues')
plt.show()
```



[32]:
```
data = {k: v/(num_of_parallel_queues * num_of_time_steps) for k, v in d.items()␣
 ↪if k <= 50}

keys = list(data.keys())
values = list(data.values())

plt.figure()
plt.bar(keys, values, color='skyblue', edgecolor='black')
plt.title('Fraction of Time Steps for Each Count Value (0 to 50)')
plt.xlabel('Number of Packets in Buffer')
plt.ylabel('Fraction of Time Steps')
plt.show()
```



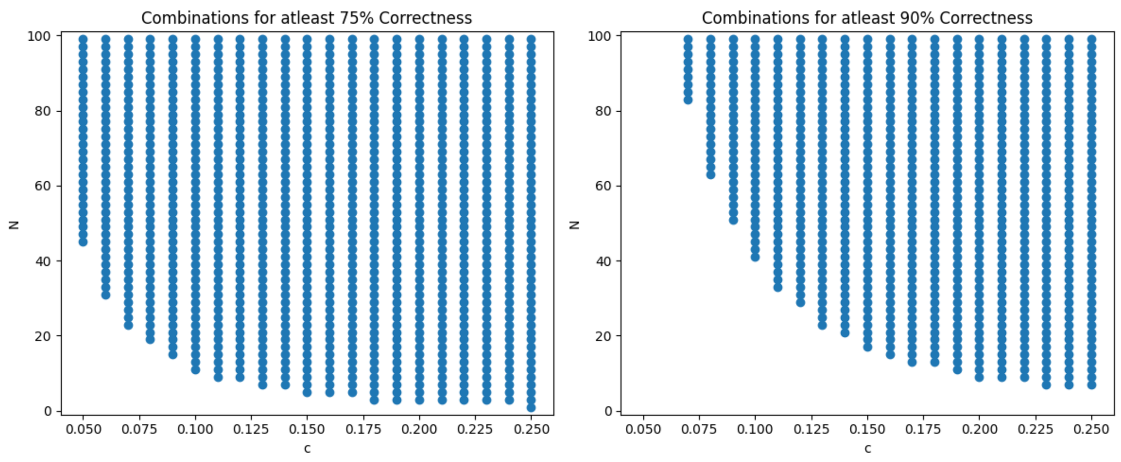
1.4 Question - 4

[33]:
```python
def binom_pmf(n, k, p):
    nck = math.factorial(n) // (math.factorial(k) * math.factorial(n - k))
    return (nck) * (p ** k) * ((1 - p) ** (n - k))

# P(X > N//2)
def major_prob(n, p):
    prob = 0
    count = (n // 2) + 1
    for k in range(count, n + 1):
        prob += binom_pmf(n, k, p)
    return prob
```

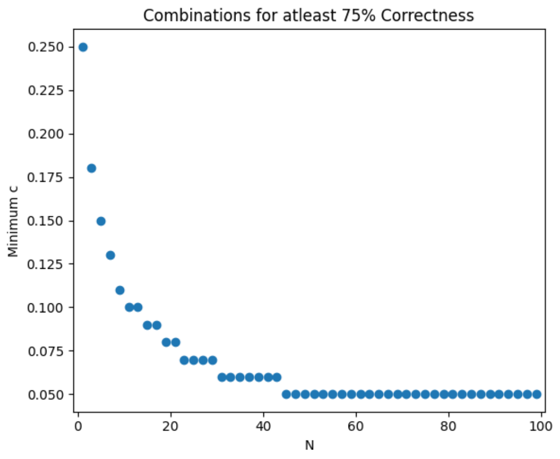
[34]:
```python
c_values = np.arange(0.05, 0.26, 0.01)
N_values = np.arange(1, 101, 2)

results_75 = []
results_90 = []
```

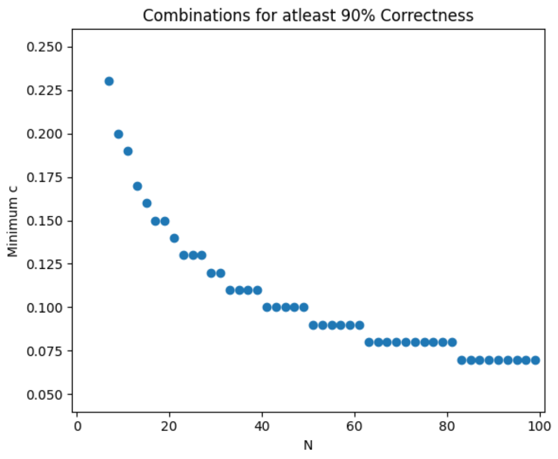
```python
min_results_75 = []
min_results_90 = []

for c in c_values:
    p = 0.5 + c
    for n in N_values:
        correct_prob = major_prob(n, p)
        if correct_prob >= 0.75:
            results_75.append((n, c))
        if correct_prob >= 0.90:
            results_90.append((n, c))

for N in N_values:
    min_75 = False
    min_90 = False
    for c in c_values:
        p = 0.5 + c
        correct_prob = major_prob(N, p)
        if not min_75 and correct_prob >= 0.75:
            min_results_75.append((N, c))
            min_75 = True
        if not min_90 and correct_prob >= 0.90:
            min_results_90.append((N, c))
            min_90 = True
        if min_75 and min_90:
            break

results_75 = np.array(results_75)
results_90 = np.array(results_90)

min_results_75 = np.array(min_results_75)
min_results_90 = np.array(min_results_90)
```

[35]:
```python
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(results_75[:, 1], results_75[:, 0])
plt.title('Combinations for atleast 75% Correctness')
plt.xlabel('c')
plt.ylabel('N')
plt.ylim(-1, 101)
plt.xlim(0.04, 0.26)

plt.subplot(1, 2, 2)
plt.scatter(results_90[:, 1], results_90[:, 0])
plt.title('Combinations for atleast 90% Correctness')
plt.xlabel('c')
plt.ylabel('N')
plt.ylim(-1, 101)
plt.xlim(0.04, 0.26)

plt.tight_layout()
plt.show()
```



[38]:
```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(min_results_75[:, 0], min_results_75[:, 1])
plt.title('Combinations for atleast 75% Correctness')
plt.xlabel('N')
plt.ylabel('Minimum c')
plt.xlim(-1, 101)
plt.ylim(0.04, 0.26)

plt.subplot(1, 2, 2)
plt.scatter(min_results_90[:, 0], min_results_90[:, 1])
plt.title('Combinations for atleast 90% Correctness')
plt.xlabel('N')
plt.ylabel('Minimum c')
plt.xlim(-1, 101)
plt.ylim(0.04, 0.26)

plt.tight_layout()
plt.show()
```





[37]:
```
def cost_function(N, c, m, n, k):
    return m * N**2 + n * np.exp(k * c)

m = 0.1
n = 10
k = 20

min_cost_75 = 100000000
optimal_75 = ()
for N, c in results_75:
    cost = cost_function(N, c, m, n, k)
    if cost < min_cost_75:
        min_cost_75 = cost
        optimal_75 = (c, N)

min_cost_90 = 100000000
optimal_90 = ()
for N, c in results_90:
    cost = cost_function(N, c, m, n, k)
    if cost < min_cost_90:
        min_cost_90 = cost
        optimal_90 = (c, N)

print(f"Optimal combination for >= 75% correctness: c = {optimal_75[0]:.2f}, N␣
 ↪= {optimal_75[1]}, Minimum Cost = {min_cost_75:.2f}")
print(f"Optimal combination for >= 90% correctness: c = {optimal_90[0]:.2f}, N␣
 ↪= {optimal_90[1]}, Minimum Cost = {min_cost_90:.2f}")
```

```
Optimal combination for >= 75% correctness: c = 0.09, N = 15.0, Minimum Cost =
83.00
Optimal combination for >= 90% correctness: c = 0.13, N = 23.0, Minimum Cost =
187.54
```