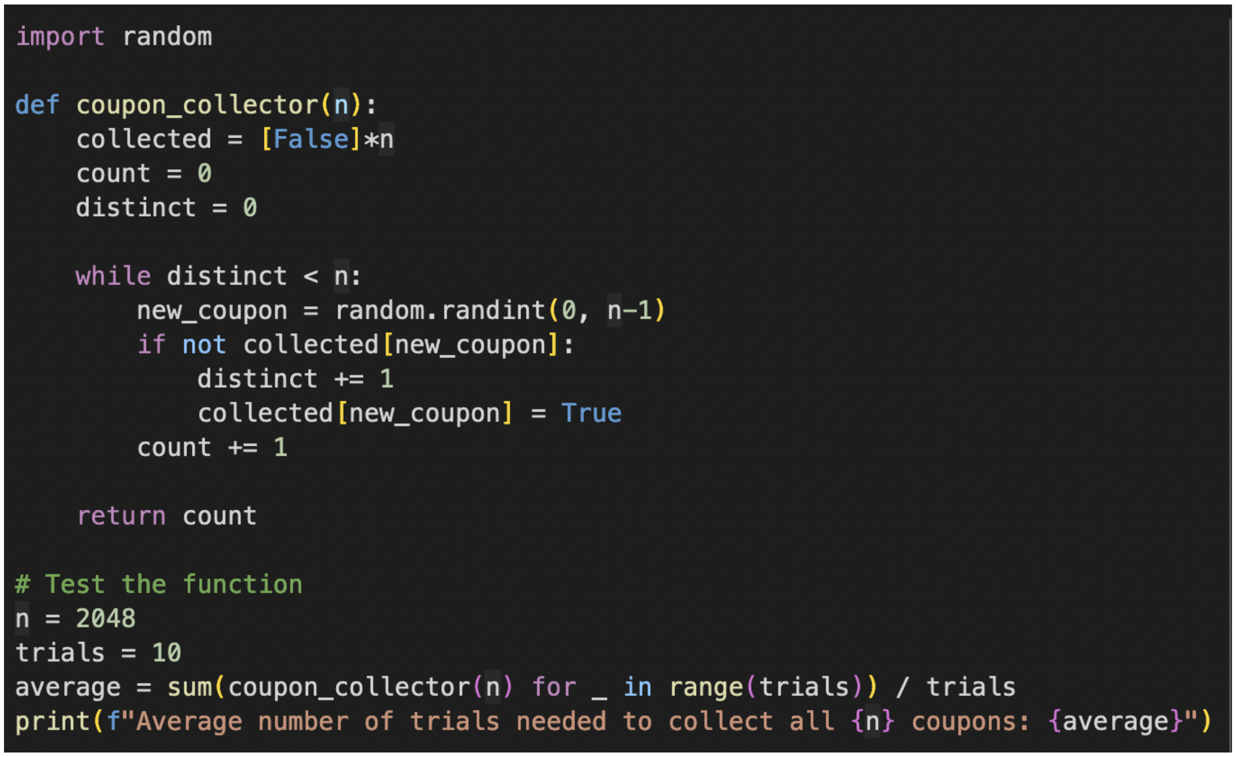
Coupon Collector's Problem

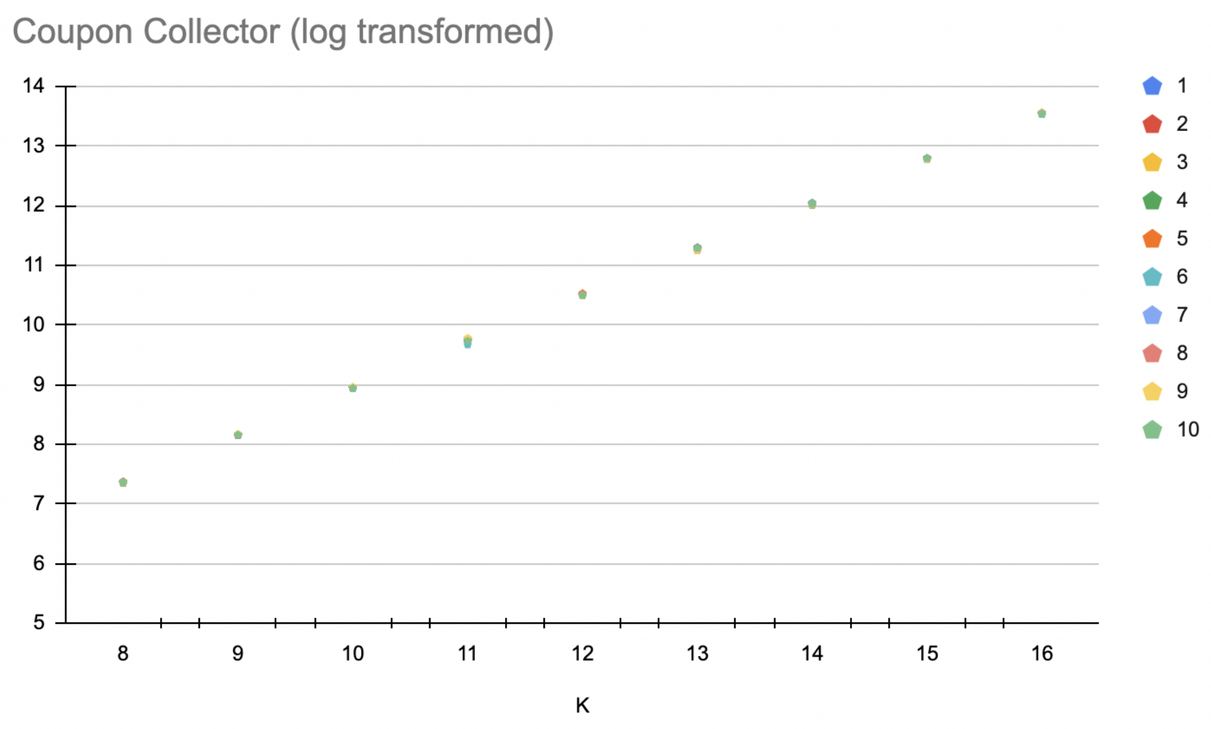


```python
import random

def coupon_collector(n):
    collected = [False]*n
    count = 0
    distinct = 0

    while distinct < n:
        new_coupon = random.randint(0, n-1)
        if not collected[new_coupon]:
            distinct += 1
            collected[new_coupon] = True
        count += 1

    return count

# Test the function
n = 2048
trials = 10
average = sum(coupon_collector(n) for _ in range(trials)) / trials
print(f"Average number of trials needed to collect all {n} coupons: {average}")
```

For a situation of n distinct tickets, our function "coupon_collector" initiates a list "collected" which is populated with n false values. It has two counter variables: "count" which keeps track of how many trials have been run thus far, and "distinct" which keeps track of how many of the distinct tickets have been collected thus far. Each time a coupon is collected, it is independently chosen at random from a set of n different types. This randomness in selection means that you cannot predict which coupon you will get in any given trial.

While the value of "distinct" is less than the n total tickets, the code has a chance of collecting a distinct ticket. If this distinct ticket has not been logged, then we collect a new ticket and thus add one to our "distinct" count, setting its corresponding numbered entry of "collected" as true. We then add one to our "count" of how many trials have run. This continues until all entries of "collected" are set to true, at which point our function returns "count".

To test our code, we ran our code with small values of n, such as $n = 5, 6, 7$ and verified that all distinct coupons were collected by having the code print "str(new_coupon)+'collected'" whenever it set the flag of a distinct coupon to "true."



Hypothesis: We hypothesize that the complexity of this data's function is O(nlogn). We log-transformed the response variable to confirm this visually.

Finding the expectation

U_i is the number of additional trials needed to collect the i-th new coupon after $i-1$ coupons have already been collected. The total number of trials T_n to collect all n coupons is then the sum of all individual U_i's:

$$T_n = U_1 + U_2 + \cdots + U_n$$

To calculate $E[U_i]$ we can use the fundamental bridge, solving for the probability of U_i. The probability of collecting a new coupon when $i-1$ coupons have already been collected is as follows where n is the total number of coupon types, and $n-(i-1)$ is the number of coupons not yet collected:

$$E[U_i] = \frac{n}{n-(i-1)}$$

The expected total number of trials $E[T_n]$ can now be calculated by summing the expected values of all U_i using linearity of expectation:

$$E[T_n] = E[U_1] + E[U_2] + \cdots + E[U_n]$$

We insert the values of $E[U_i]$ (using the probability) we calculated above:

$$E[T_n] = 1 + \frac{n}{n-1} + \frac{n}{n-2} + \cdots + \frac{n}{1}$$

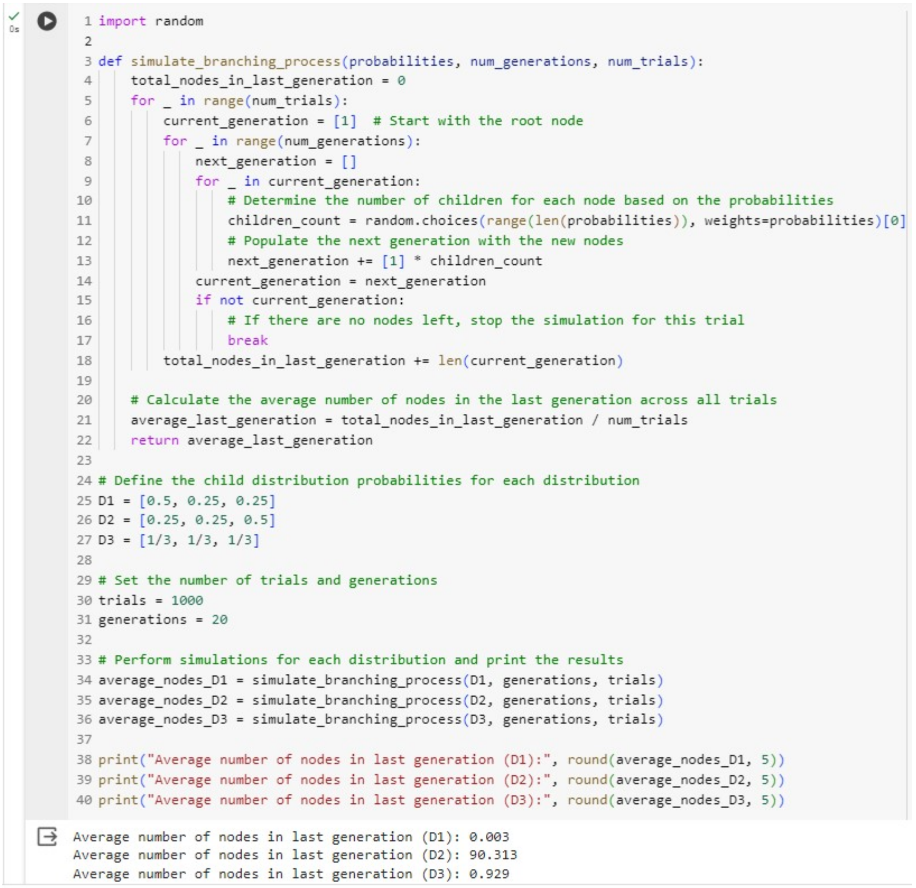
We can factor out the n to help simplify the value of $E[T_n]$:

$$E[T_n] = n\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}\right)$$

This model can be expressed as the harmonic number which equates $n\log(n)$.

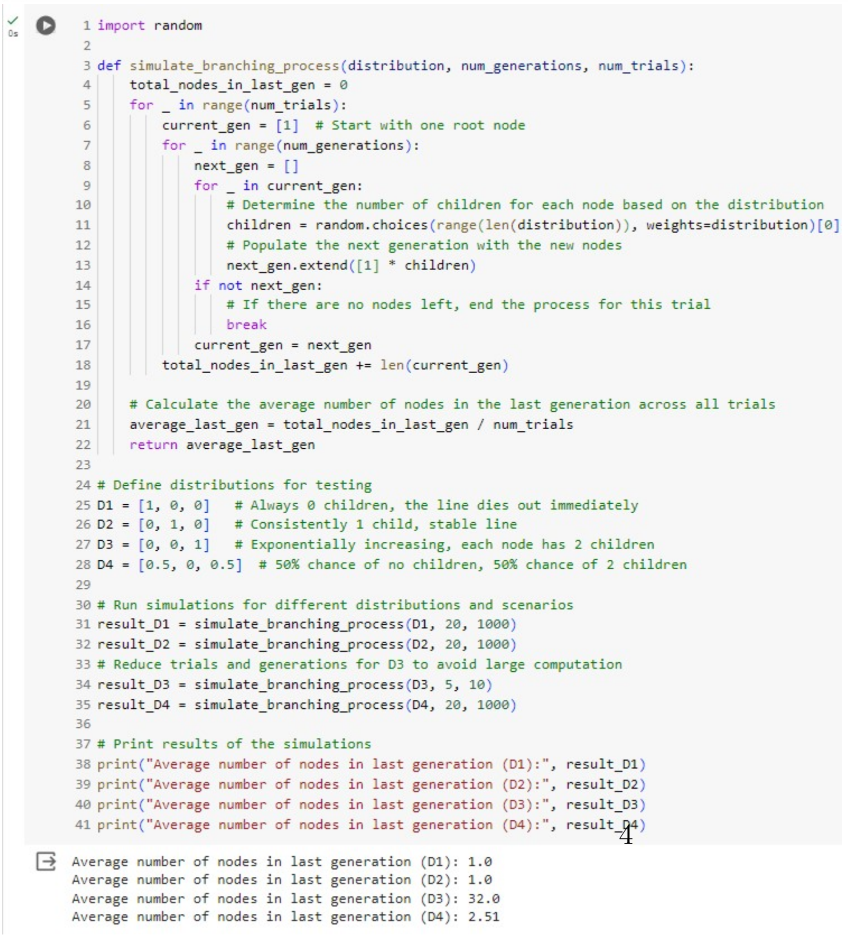
Our hypothesis and experimental results seem to align with the formal calculations as they equate to the same mathematical expression.

Code Simulation



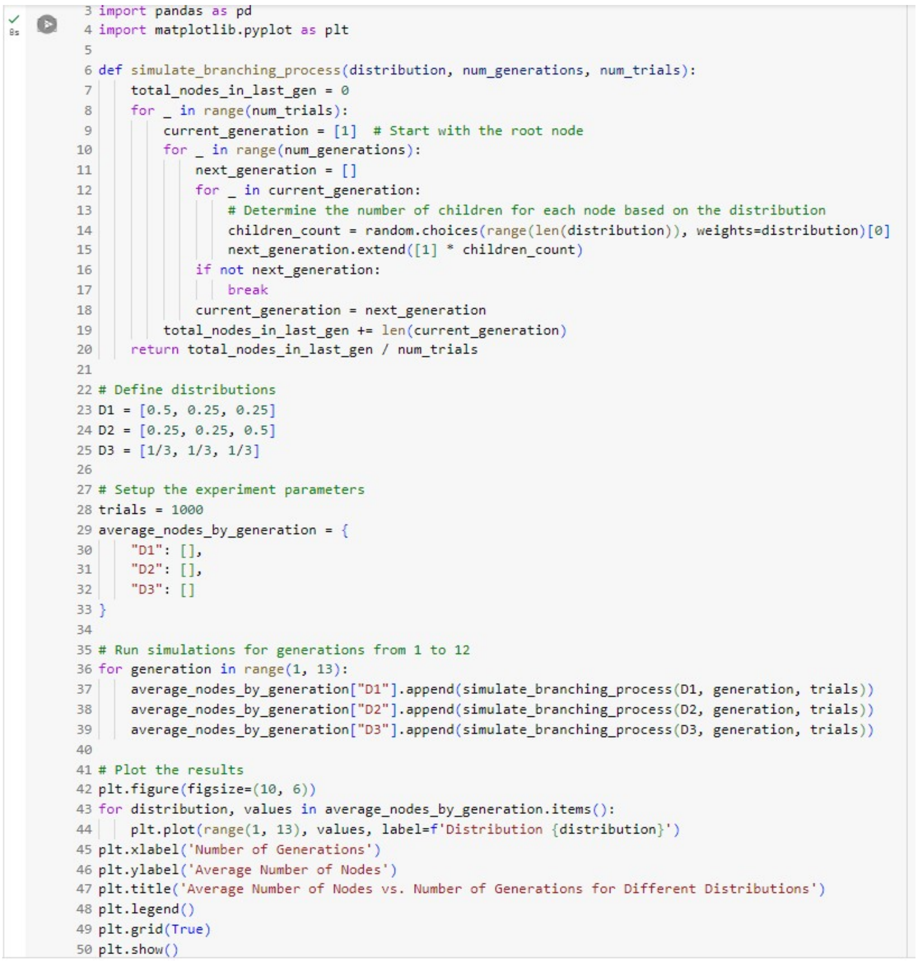
```python
import random

def simulate_branching_process(probabilities, num_generations, num_trials):
    total_nodes_in_last_generation = 0
    for _ in range(num_trials):
        current_generation = [1]  # Start with the root node
        for _ in range(num_generations):
            next_generation = []
            for _ in current_generation:
                # Determine the number of children for each node based on the probabilities
                children_count = random.choices(range(len(probabilities)), weights=probabilities)[0]
                # Populate the next generation with the new nodes
                next_generation += [1] * children_count
            current_generation = next_generation
            if not current_generation:
                # If there are no nodes left, stop the simulation for this trial
                break
        total_nodes_in_last_generation += len(current_generation)

    # Calculate the average number of nodes in the last generation across all trials
    average_last_generation = total_nodes_in_last_generation / num_trials
    return average_last_generation

# Define the child distribution probabilities for each distribution
D1 = [0.5, 0.25, 0.25]
D2 = [0.25, 0.25, 0.5]
D3 = [1/3, 1/3, 1/3]

# Set the number of trials and generations
trials = 1000
generations = 20

# Perform simulations for each distribution and print the results
average_nodes_D1 = simulate_branching_process(D1, generations, trials)
average_nodes_D2 = simulate_branching_process(D2, generations, trials)
average_nodes_D3 = simulate_branching_process(D3, generations, trials)

print("Average number of nodes in last generation (D1):", round(average_nodes_D1, 5))
print("Average number of nodes in last generation (D2):", round(average_nodes_D2, 5))
print("Average number of nodes in last generation (D3):", round(average_nodes_D3, 5))
```

```
Average number of nodes in last generation (D1): 0.003
Average number of nodes in last generation (D2): 90.313
Average number of nodes in last generation (D3): 0.929
```

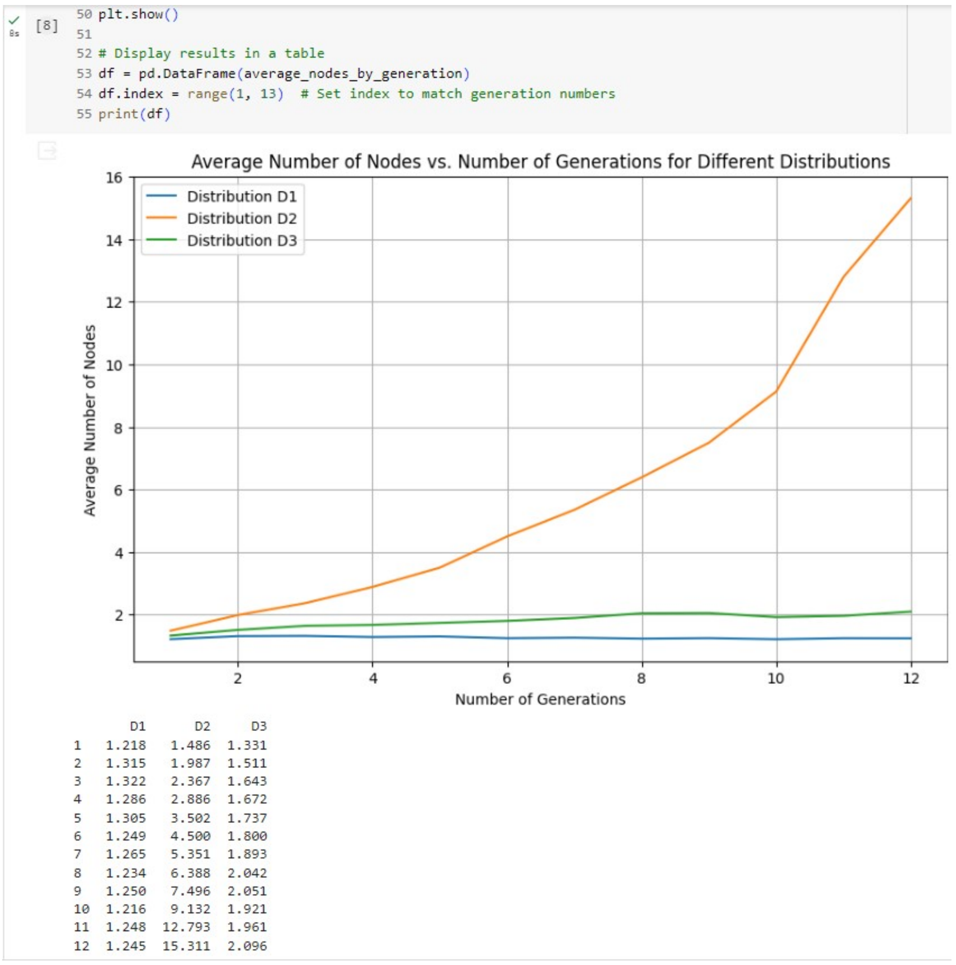


```python
import random

def simulate_branching_process(distribution, num_generations, num_trials):
    total_nodes_in_last_gen = 0
    for _ in range(num_trials):
        current_gen = [1]  # Start with one root node
        for _ in range(num_generations):
            next_gen = []
            for _ in current_gen:
                # Determine the number of children for each node based on the distribution
                children = random.choices(range(len(distribution)), weights=distribution)[0]
                # Populate the next generation with the new nodes
                next_gen.extend([1] * children)
            if not next_gen:
                # If there are no nodes left, end the process for this trial
                break
            current_gen = next_gen
        total_nodes_in_last_gen += len(current_gen)

    # Calculate the average number of nodes in the last generation across all trials
    average_last_gen = total_nodes_in_last_gen / num_trials
    return average_last_gen

# Define distributions for testing
D1 = [1, 0, 0]   # Always 0 children, the line dies out immediately
D2 = [0, 1, 0]   # Consistently 1 child, stable line
D3 = [0, 0, 1]   # Exponentially increasing, each node has 2 children
D4 = [0.5, 0, 0.5]  # 50% chance of no children, 50% chance of 2 children

# Run simulations for different distributions and scenarios
result_D1 = simulate_branching_process(D1, 20, 1000)
result_D2 = simulate_branching_process(D2, 20, 1000)
# Reduce trials and generations for D3 to avoid large computation
result_D3 = simulate_branching_process(D3, 5, 10)
result_D4 = simulate_branching_process(D4, 20, 1000)

# Print results of the simulations
print("Average number of nodes in last generation (D1):", result_D1)
print("Average number of nodes in last generation (D2):", result_D2)
print("Average number of nodes in last generation (D3):", result_D3)
print("Average number of nodes in last generation (D4):", result_D4)
```

```
Average number of nodes in last generation (D1): 1.0
Average number of nodes in last generation (D2): 1.0
Average number of nodes in last generation (D3): 32.0
Average number of nodes in last generation (D4): 2.51
```



```python
import pandas as pd
import matplotlib.pyplot as plt

def simulate_branching_process(distribution, num_generations, num_trials):
    total_nodes_in_last_gen = 0
    for _ in range(num_trials):
        current_generation = [1]  # Start with the root node
        for _ in range(num_generations):
            next_generation = []
            for _ in current_generation:
                # Determine the number of children for each node based on the distribution
                children_count = random.choices(range(len(distribution)), weights=distribution)[0]
                next_generation.extend([1] * children_count)
            if not next_generation:
                break
            current_generation = next_generation
        total_nodes_in_last_gen += len(current_generation)
    return total_nodes_in_last_gen / num_trials

# Define distributions
D1 = [0.5, 0.25, 0.25]
D2 = [0.25, 0.25, 0.5]
D3 = [1/3, 1/3, 1/3]

# Setup the experiment parameters
trials = 1000
average_nodes_by_generation = {
    "D1": [],
    "D2": [],
    "D3": []
}

# Run simulations for generations from 1 to 12
for generation in range(1, 13):
    average_nodes_by_generation["D1"].append(simulate_branching_process(D1, generation, trials))
    average_nodes_by_generation["D2"].append(simulate_branching_process(D2, generation, trials))
    average_nodes_by_generation["D3"].append(simulate_branching_process(D3, generation, trials))

# Plot the results
plt.figure(figsize=(10, 6))
for distribution, values in average_nodes_by_generation.items():
    plt.plot(range(1, 13), values, label=f'Distribution {distribution}')
plt.xlabel('Number of Generations')
plt.ylabel('Average Number of Nodes')
plt.title('Average Number of Nodes vs. Number of Generations for Different Distributions')
plt.legend()
plt.grid(True)
plt.show()
```



```python
plt.show()

# Display results in a table
df = pd.DataFrame(average_nodes_by_generation)
df.index = range(1, 13)  # Set index to match generation numbers
print(df)
```

```
       D1      D2     D3
1   1.218   1.486  1.331
2   1.315   1.987  1.511
3   1.322   2.367  1.643
4   1.286   2.886  1.672
5   1.305   3.502  1.737
6   1.249   4.500  1.800
7   1.265   5.351  1.893
8   1.234   6.388  2.042
9   1.250   7.496  2.051
10  1.216   9.132  1.921
11  1.248  12.793  1.961
12  1.245  15.311  2.096
```

The simulation starts by initializing a list called currentgeneration with one element, [1]. This represents the root node. For each trial, the simulation goes through the desired number of genertions. A new list called nextgeneration is created to store nodes for the upcoming generation. During each generation, every node in the current list is processed to determine how many children it will have based on the probability distribution D (using random.choices method). This method returns a list of selections according to the weights in D, and only the first item which indicates however many children there are for that node, is used. If a node has children, these are added to nextgeneration. If nextgeneration is empty, indicating that no nodes were generated and the lineage has died out in that trial, the generation process stops. We then count the number nodes in the final generation of each trial. After all trials are complete, this total is divided by the number of trials to compute the average amount of nodes in the last generation.

The simulation was tested under diffent conditions for experimental rigor. One test ensured that when each node produces no children, the result is zero nodes in all subsequent generations to make sure it worked as expected. There was another test where each node consistently produced exactly one child confirmed that each generation contained precisely one node. In a test designed to observe exponential growth, where each node produced two children, the results matched what we expected to see, especially in the fifth generation with 32 nodes. Additionally, a scenario was tested where nodes had an equal chance of producing zero or two children. The average outcome of this setup was close to one child per node, with results showing an average of approximately 1.126

nodes, demonstrating the simulation's accuracy under probabilistically varied conditions. These tests show how robust the simulation was across different hypothetical scenarios.

Expectation analysis

If we assume Y_j is the number of children that the j^{th} node in the n^{th} generation has, then by linearity of (conditional expectation), $E(X_{n+1}|X_n = k) = E(Y_1 + \cdots + Y_k|X_n = k) = \sum_{j=1}^{k} E(Y_j|X_n = k) = \sum_{j=1}^{k} E(Y_j)$ because Y_j is independent of $X_n = k$. This in turn is equal to $kE(Y_j)$ by symmetry of Y_js).

We break it down for D_1, D_2, and D_3, respectively:
$E(Y_j) = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 2 \cdot \frac{1}{4} = \frac{3}{4}$. Thus, $E(X_{n+1}|X_n = k) = \frac{3k}{4}$
$E(Y_j) = 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{4} + 2 \cdot \frac{1}{2} = \frac{5}{4}$. Thus, $E(X_{n+1}|X_n = k) = \frac{5k}{4}$
$E(Y_j) = 0 \cdot \frac{1}{3} + 1 \cdot \frac{1}{3} + 2 \cdot \frac{1}{3} = 1$. Thus, $E(X_{n+1}|X_n = k) = k$

Proof of Hypothesis by Induction

Claim: We will let $P(n)$ be the proposition that when we start with a single root node to start the branching process, $E(X_n) = \mu^n$, where μ is the mean of distribution D.

Base Case, $P(1)$: $X_0 = 1$ since the 0^{th} generation is just the root node itself. So the distribution of X_1 is the distribution of D. This implies that $E(X_1) = \mu = \mu^1$. Thus we have proven $P(1)$ to be true.

Inductive Hypothesis: We will assume that $P(m)$ is true, letting $m \in \mathbb{Z}^+$ be arbitrarily chosen. So, $E(X_m) = \mu^m$.

Inductive Step: By the inductive hypothesis, $E(X_{m+1}) = E(Y_1 + \ldots + Y_m) = \mu^m E(Y_1)$.

Y_j represents the number of children that the j^{th} node in the m^{th} generation has, and $j = 1, 2, \ldots, m$ by the distribution D, so $E(Y_1) = \mu$.

Hence, $E(X_{m+1}) = \mu^m \cdot \mu = \mu^{m+1}$ and $P(m) \Rightarrow P(m + 1)$ for all $m \in \mathbb{Z}^+$.

Because $P(1)$ is true and $P(m) \Rightarrow P(m + 1)$ for all $m \in \mathbb{Z}^+$, then by mathematical induction, $E(X_n) = \mu^n$ where μ is the mean of distribution D.

Analysis

- **For D_1:** It was determined that the average number of nodes in the 20^{th} generation is 0.008. This is reasonably close to the true value, $\left(\frac{3}{4}\right)^{20} \approx 0.0031$
- **For D_2:** Experiments predicted $89.5\ldots$, which was reasonably close to the true value of $\left(\frac{5}{4}\right)^{20} \approx 86.7$
- **For D_3:** Experiments predicted 0.944, which is was reasonably close to the true value of 1.