Problem 1. (Limits of Binomial Distributions)

This problem justifies the claim that the sparse $G(n, p)$ model has Poisson degree distributions as n grows large.

As a reminder, a nonnegative discrete random variable X has a *binomial distribution* with number of trials n and success rate p if its probability mass function is

$$p_X(k) = \binom{n}{k} p^k (1-p)^{n-k} \; .$$

A nonnegative discrete random variable Y has a *Poisson distribution* with parameter μ if its probability mass function is

$$p_Y(k) = \frac{\mu^k e^{-\mu}}{k!} \; .$$

Suppose now that X is binomial and that $p = \frac{c}{n}$.[1] Show that, as n grows large, X is distributed approximately Poisson. Identify the parameter of the Poisson distribution in terms of c and n. It suffices to show that the PMF of X converges to a Poisson PMF (formally, we are showing *convergence in distribution*).

An argument with calculations to back it up, rather than a formal proof, is sufficient. You may need to find some useful approximations for quantities like $\binom{n}{k}$ and e^m. You're welcome to use any approximations you can find, as long as you **cite your sources**.

Let us first substitute $p = \frac{c}{n}$:

$$P_X(k) = \binom{n}{k} (\frac{c}{n})^k (1 - \frac{c}{n})^{n-k}$$

Then we use the following result from this Math Stack Exchange post for an approximation $\binom{n}{k} \approx \frac{n^k}{k!}$ as $n \to \infty$:

$$\approx (\frac{n^k}{k!})(\frac{c}{n})^k (1 - \frac{c}{n})^{n-k} \text{ as } n \to \infty$$

$$= (\frac{c^k}{k!})(1 - \frac{c}{n})^{n-k} \text{ as } n \to \infty$$

We then use the following from UVA lecture notes on the exponential function, which gives the rewritten definition as: $\lim_{n\to\infty}(1 + \frac{c}{n})^n = e^c$, such that $\lim_{n\to\infty}(1 - \frac{c}{n})^n = e^{-c}$:

$$\lim_{n\to\infty} P_X(k) = (\frac{c^k}{k!})e^{-c}$$

Notice that given the Poisson definition $P_Y(k) = \frac{\mu^k e^{-\mu}}{k!}$, for $\mu = c$, these expressions are equivalent; $\lim_{n\to\infty} P_X(k) = P_Y(k)$.

[1] In class we considered $p = \frac{c}{n-1}$; the choice $p = \frac{c}{n}$ is more convenient and equivalent as n grows large.

Problem 2. (Generating Chung-Lu Graphs I)

In a previous homework problem, we considered an alternative algorithm for generating an Erdős-Rényi random graph which did not require us to flip a coin for each of the $n(n-1)/2$ pairs of nodes in the graph. Instead, we generated a binomially-generated number of edges M and then distributed those edges amongst the pairs of nodes in a way that led to the same distribution of edges as in the Erdős-Rényi. It is not necessary to complete that problem in order to complete this one, but it does help.

Consider the following algorithm for generating a Chung-Lu random graph. We'll consider a version of the Chung-Lu random graph in which the number of edges between nodes i and j is Poisson-distributed with mean $\frac{k_i k_j}{2m}$, with self-loops and multiedges allowed. Normally, sampling from this model would require that we consider each of $n^2/2$ possible pairs of nodes (including self-loops) and sample from a Poisson distribution for each. Can we do this using fewer random numbers? Here's an approach:

i. First, sample M edges from a Poisson distribution with mean x.

ii. Then, distribute each of these M edges amongst the $n^2/2$ possible pairs of nodes (including self-loops) so that the expected number of edges between nodes i and j is proportional to $k_i k_j$.

Part (a)

Determine the appropriate value of x in terms of the expected degree sequence **k**. It is sufficient to ensure that the total number of edges generated is correct in expectation.

Part (b)

Show that the number of edges between nodes i and j under this algorithm is indeed exactly Poisson-distributed with mean $\frac{k_i k_j}{2m}$.

Hint: You may find the *Poisson splitting property* to be helpful: if I sample X from a Poisson distribution with mean μ and then flip X coins with success probability p, the number of successful coin flips is Poisson-distributed with mean μp. You may use this property without proof.

P2) Chung Lu random graph generation:

- # edges between i, j is poisson distributed w/ mean $\frac{k_i k_j}{2m}$ w/ self loops and multiedges allowed.
- This would normally require considering $\frac{n^2}{2}$ pairs of nodes, sampling from Poisson for each.

Consider:

1) sample M edges from Poisson w/ mean x.
2) distribute each of M edges amongst $n^2/2$ possible pairs of nodes st $\mathbb{E}$ of # edges b/w i and j is proportional to $k_i k_j$

a) determine appropriate value of x in terms of degree sequence k.

#edges b/w i, j is Poisson distributed st $\mathbb{E}[\text{\# edges between } i, j] = \left[\frac{k_i k_j}{2m}\right]$ = x

Thus over all i, j:

$$\mathbb{E}[\text{total \# of edges}] = \frac{1}{2}\sum_{i,j} \mathbb{E}[\text{\# edges b/w } i,j] = \frac{1}{2}\sum_{i,j} \frac{k_i k_j}{2m}$$

$$= \frac{1}{4m}\sum_{i,j} k_i k_j$$, rewriting as $\frac{1}{4m}\left(\sum_i k_i\right)^2$, given $\sum_{i,j} k_i k_j = \sum_i k_i \sum_j k_j$

notice that the sum over all k_i is simply 2m, since we count ends of each edge twice for given node degrees. Thus:

$$= \frac{1}{4m}(2m)^2 = m$$. We showed it is sufficient to ensure # edges = m in expectation

b) Show that the number of edges b/w i and j is exactly Poisson distributed with mean $\frac{K_i K_j}{2m}$

prove E[# edges i and j] is Poisson distributed w/ mean $\frac{K_i K_j}{2m}$

Let $M \sim$ Poisson (m) be the number of sampled edges.

Then each M is distributed over $\frac{n^2}{2}$ possible pairs of nodes s.t.

expected edges i to j $\propto K_i K_j$ (by the algorithm defn)

Over all (i,j): $\frac{1}{2}\sum_{i,j} K_i K_j = \frac{1}{2}\left(\sum_i K_i\right)^2 = \frac{1}{2}(2m)^2$

So, $p = \frac{K_i K_j}{\frac{1}{2}(2m)^2}$ in order to account for proportion over all i,j pairs

by the Poisson Splitting Property, we assign events of M independently to (i,j)

so, # of edges between i and j $\sim$ Poisson $(m \cdot p)$

rewritten as Poisson$\left(\frac{m\, K_i K_j}{\frac{1}{2}(2m)^2}\right)$

thus [# of edges between i and j] $\sim$ Poisson $\left(\frac{K_i K_j}{2m}\right)$

Problem 3. (Generating Chung-Lu Graphs II, 2 points)

We'll consider the Poisson Chung-Lu model with self-loops and multiedges that we introduced in the previous problem.

In this problem, we are actually going to test the speed of two competing implementations for generating large Chung-Lu random graphs. Because NetworkX itself is slow and we don't want to just benchmark NetworkX, your output in each of the algorithms below should be an edge list: a list of pairs of nodes representing edges in the graph.

Part (a)

Please implement the alternative algorithm for sampling from the Chung-Lu model that you outlined in the previous part. In your implementation, please find a way to avoid having to explicitly form an array containing $n^2/2$ probabilities for where to place each edge; instead, see if you can work only with arrays of size n. As specified above, your output should be an edge list.

Part (b)

Please also implement a version of the Poisson Chung-Lu model that allows for self-loops and multiedges using the "regular" algorithm as outlined in the lecture notes. As specified above, your output should be an edge list.

Part (c)

Let's now generate Chung-Lu random graphs as follows. First, we'll fix n. Then, for each $i = 1, \ldots, n$, we'll sample k_i uniformly from the set $\{1, \ldots, 10\}$. Finally, we'll construct a Chung-Lu random graph with n nodes and degree sequence $\mathbf{k}$. We're going to allow n to vary; since we are fixing the distribution of k_i this corresponds to a version of sparse Chung-Lu.

Please perform this task using both of the algorithms that you implemented in this problem for varying n. Use any method to measure the *time* (in seconds) required to generate your edge lists. Let n range from 10 to 1000 and perform at least 10 trials for each value of n. Plot the resulting estimates. Which algorithm would you recommend for generating sparse Chung-Lu random graphs when n is large?

Problem 3

Part A

In [2]:
```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import random

def AlternativeChungLu(degs):
    edge_list = []
    K = sum(degs)
    M = np.random.poisson(K/2)
    n = len(degs)
    weights = weights=degs/sum(degs)

    for _ in range(M):
        i = random.choices(range(n), weights=weights, k=1)[0]
        j = random.choices(range(n), weights=weights, k=1)[0]
        edge_list.append((i, j))
    return edge_list
```

Part B

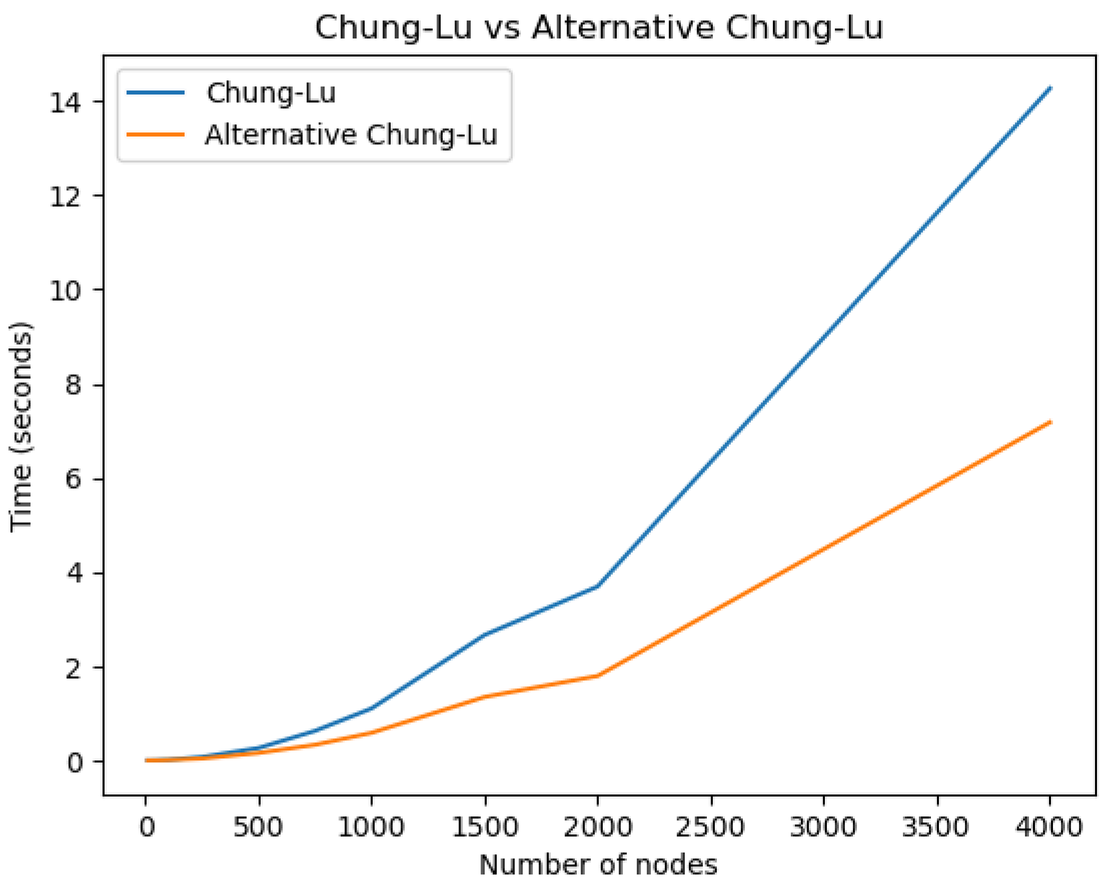
In [2]:
```
def ChungLu(degs):
    n = len(degs)
    edge_list = []
    deg_sum = sum(degs)

    for i in range(n):
        for j in range(i, n):
            mean_edges = (degs[i] * degs[j]) / deg_sum
            num_edges = np.random.poisson(mean_edges)

            for _ in range(num_edges):
                edge_list.append((i, j))
    return edge_list
```

Part C

In [3]:
```
import time

ns = [10, 15, 25, 50, 75, 100, 250, 500, 750, 1000, 1500, 2000, 4000]
CLTimes = []
ACLTimes = []

for n in ns:
    in_degs = np.random.randint(1, 10, size=n)

    start = time.time()
    ChungLu(in_degs)
    end = time.time()
    CLTimes.append(end - start)

    start = time.time()
    AlternativeChungLu(in_degs)
    end = time.time()
    ACLTimes.append(end - start)
```

In [4]:
```
plt.plot(ns, CLTimes, label='Chung-Lu')
plt.plot(ns, ACLTimes, label='Alternative Chung-Lu')
plt.xlabel('Number of nodes')
plt.ylabel('Time (seconds)')
plt.title('Chung-Lu vs Alternative Chung-Lu')
plt.legend()
```

Out[4]:
```
<matplotlib.legend.Legend at 0x1c9bad61940>
```



In [ ]:

The Alternative Chung-Lu algorithm seems like a much faster method for very large n.

Problem 4. (Configuration Model Properties)

Consider a configuration-model random graph with degree sequence $\{k_1, ..., k_n\}$, constructed via stub-matching. Unlike in the lecture notes, we'll allow this algorithm to generate both self-loops and multiedges. Define

$$\mathbb{P}_{ij} \triangleq \frac{k_i k_j}{2m}\,.$$

In lecture, we argued that $\mathbb{P}_{ij}$ is a reasonable approximation for the expected number of edges between i and j.

i. Calculate the probability that there exist (at least) two edges between nodes i and j.

ii. Using the previous part, show that the expected number of pairs of nodes with two or more edges between them is approximately

$$\frac{1}{2}\left[\frac{\langle k^2\rangle - \langle k\rangle}{\langle k\rangle}\right]^2\,,$$

where $\langle k\rangle = \frac{1}{n}\sum_i k_i$ and $\langle k^2\rangle = \frac{1}{n}\sum_i k_i^2$.

iii. Show that the expected number of nodes with self-loops is approximately

$$\frac{1}{2}\frac{\langle k^2\rangle - \langle k\rangle}{\langle k\rangle}\,. \tag{1}$$

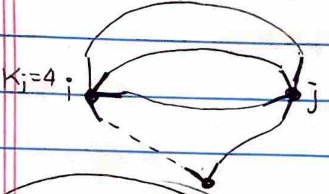
iv. Show that the expected number of common neighbors of two nodes i and j is approximately

$$\mathbb{P}_{ij}\frac{\langle k^2\rangle - \langle k\rangle}{\langle k\rangle}\,.$$

P4) Consider a configuration model random graph w/ $K = \{K_1, \ldots, K_n\}$ constructed via stub matching. We allow self loops/ multiedges.

$P_{i,j} \triangleq \frac{K_i K_j}{2m}$, a reasonable approximation of edges between i, j

• i) Calculate $\mathbb{P}($ 2 or more edges between i, j$)$



$\binom{K_i}{2}$ ← ways of choosing 2 stubs that are distinct

× $\mathbb{P}($stub #1 on node i connects to any K_j stubs of j$)$

× $\mathbb{P}($stub #2 on node i connects to any $K_j - 1$ stub of j$)$

→ this is just $\frac{K_j}{2m-1}$

→ $\frac{K_j - 1}{2m-3}$ ← 1 fewer stubs ← 2 fewer stub ends overall after first edge (i,j)

$$\boxed{= \binom{K_i}{2} \frac{K_j(K_j-1)}{(2m-1)(2m-3)}}$$

• ii) show that the expected # of pair of nodes w/ 2+ edges is approx

$$\frac{1}{2}\left[\frac{\langle K^2\rangle - \langle K\rangle}{\langle K\rangle}\right]^2$$

$$\mathbb{E}[\text{\# nodes pairs w/2+ edges between}] = \sum_{i,j} \mathbb{P}(\text{i, j share 2+ edges})$$

$$= \sum_{i,j} \frac{K_i(K_i-1)}{2} \cdot \frac{K_j(K_j-1)}{(2m-1)(2m-3)}$$. For large m, $2m \approx 2m-1 \approx 2m-3$

$$\approx \frac{1}{2(2m)^2} \sum_{i,j} K_i(K_i-1) \times K_j(K_j-1)$$

We know $\sum_i K_i = 2m$ and $\sum_i K_i \approx N\langle K\rangle$.

$$\approx \frac{1}{2(N\langle K\rangle)^2} (N\langle K^2\rangle - N\langle K\rangle)(N\langle K^2\rangle - N\langle K\rangle)$$

$$= \frac{1}{2} \frac{(\langle K^2\rangle - \langle K\rangle)^2}{\langle K\rangle^2} = \frac{1}{2}\left[\frac{\langle K\rangle^2 - \langle K\rangle}{\langle K\rangle}\right]^2$$

• iii) Show that the expected # nodes w/ self loops is $\approx \frac{1}{2} \frac{\langle K^2\rangle - \langle K\rangle}{\langle K\rangle}$

$$\mathbb{E}[\text{\# nodes w/ self loops}] = \sum_i \mathbb{P}(\text{node i has self loop})$$

$\binom{K_i}{2} \frac{1}{2m-1}$

→ number of ways of choosing two stubs on the same node i

odds we choose the other stub with 2m − 1 overall stubs left to choose from

Let us rewrite, using $2m \approx 2m-1$ for large m:

$\simeq \sum_i \frac{K_i(K_i - 1)}{2} \times \frac{1}{2m}$, rewriting $\binom{K_i}{2}$ and plugging the probability in.

$\simeq \frac{1}{2(2m)} \sum_i K_i^2 - K_i$

given $2m = \sum_i K_i = N\langle K \rangle$, we can swap out $2m$

$= \boxed{\frac{1}{2} \frac{\cancel{N}\langle K^2 \rangle - \cancel{N}\langle K \rangle}{\cancel{N}\langle K \rangle}} = \frac{1}{2} \frac{\langle K^2 \rangle - \langle K \rangle}{\langle K \rangle}$

- iv) Show that the expected number of common neighbors of two nodes i and j is $\approx P_{i,j} \frac{\langle K^2 \rangle - \langle K \rangle}{\langle K \rangle}$

$\mathbb{E}$[number of common neighbors of two nodes i and j] =

(We know that in our model, $P_{i,j} \simeq \frac{K_i K_j}{2m}$)

For node i with neighbor u, there is probability $\approx \frac{K_i K_u}{2m}$ of that edge.
For a second edge on u from j, we have $K_u - 1$ remaining,
so approximately $\frac{K_j(K_u - 1)}{2m}$ in probability.

Over all neighbors possible for u we have: $\sum_u \left(\frac{K_i K_u}{2m}\right)\left(\frac{K_j(K_u - 1)}{2m}\right)$

$= \frac{K_i K_j}{(2m)^2} \sum_u (K_u^2 - K_u)$ after rearranging:

We know $\sum_u K_u^2 = N\langle K^2 \rangle$ and $\sum_u K_u = N\langle K \rangle$, and $2m = N\langle K \rangle$ $= N \sum_i K_i$

Plugging this in:

$= \frac{K_i K_j (N\langle K^2 \rangle - N\langle K \rangle)}{N^2 \langle K \rangle^2}$

Plugging in $P_{ij} \simeq \frac{K_i K_j}{2m} = \frac{K_i K_j}{N\langle K \rangle}$:

$= P_{ij} \left(\frac{\cancel{N}\langle K^2 \rangle - \cancel{N}\langle K \rangle}{\cancel{N}\langle K \rangle}\right) = P_{ij} \left(\frac{\langle K^2 \rangle - \langle K \rangle}{\langle K \rangle}\right)$

Problem 5. (How Many Configurations?)

Show that for a given degree sequence $\mathbf{k}$ such that $\sum_i k_i = 2m$, the number of possible stub-matchings (outputs of the stub-matching procedure prior to trimming self-loops and multi-edges) is

$$\frac{(2m)!}{2^m(m!)}\,.$$

Interesting, this number is independent of the degree sequence, except for the dependence on the total number of edges m.

P5) show that for given degree sequence K s.t. $\sum_i K_i = 2m$, the number of possible stub matchings is $\frac{(2m)!}{2^m(m!)}$

There are $2m$ stubs for m distinct edges, as each edge must have attachment to 2 nodes. Let configuration model graph G consist of degree sequence $K = \{K_1, K_2, \ldots K_n\}$. The sum over K gives us $2m$ stubs.
We can construct a list of nodes $N = \{1, \ldots 1, 2, \ldots, 2, n \ldots n\}$ such that the multiplicity of nodes i corresponds to degree K_i, where $|N| = 2m$.
Consequently there are $(2m)!$ ways we can arrange node/"stubs" in this list.
We can then match pairs of stubs, forming m edges. Though, the ordering of these m edges do not matter; there are $m!$ ways of ordering, so we take $\frac{(2m)!}{(m!)}$ to get the number of matchings that does NOT account for order of stubs within edges.
To account for this, we know that there are 2 ends to m edges, so 2^m different ways of ordering within the m edges.
If we then account for this ordering, we arrive at $\frac{(2m)!}{2^m m!}$ possible stub matches.

Problem 6. (Faster Preferential Attachment (2 points))

In the online lecture notes, we implemented a preferential attachment which took a degree-proportional sampling step with probability α and a uniform sampling step with probability $1-\alpha$. However, the version of the model that we implemented was very slow.

To submit this problem, I recommend working in a Jupyter Notebook and converting the result to PDF. There are several ways to achieve this, including the Quarto technical publishing system for those who are familiar as well as opening your notebook in a browser (or using Colab) and choosing Print $\to$ As PDF in your browser.

Part (a)

Implement a faster preferential attachment simulation. Your simulation should be a Python function which allows the user to specify the number of steps and the parameter α. The return value should be a NetworkX graph object. Your implementation should be fast enough that it is possible to run 10^6 preferential attachment steps within 30 seconds on your laptop. Please structure your solution as a function that accepts a desired number of steps as input and returns a NetworkX graph object.

Hint: think about your data structures. Can you avoid computing the degree vector from scratch in each timestep?

Part (b)

Fix a value of α. Make a plot comparing the runtime of your faster simulation with the runtime of the slower simulation as n grows large. You can try $n = 10^h$ for $h = 1, 2, \ldots, 4$ or so; past $n = 10^4$ will likely be prohibitive for the slower of the two implementations.

Part (c)

Simulate 10^6 steps of preferential attachment for $\alpha = 0.5$, $\alpha = 0.75$, and $\alpha = 1.0$ using your fast algorithm Show log-binned histograms of your results. Also plot a line showing the theoretically predicted slope as in the lecture notes. You are encouraged to use any code from the lecture notes with attribution.

Part A

In [ ]:
```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import time

def OptimizedPreferentialAttachment(n_steps, alpha):

    # Start w/ 2 nodes and in hypothetical scenario we add 1 node at every step, gi
    # all nodes start with degree zero for all possible nodes (except for the first
    max_nodes = n_steps + 2
    degrees = np.zeros(max_nodes, dtype=int)
    degrees[0:2] = 1

    # Hold nodes w/ multiplcity of their degree which allows uniform sampling;
    # size is safe enough? TODO CHECK: add 2 nodes per step is max 2*n_steps + 2?
    repeated_nodes = np.zeros(2 *max_nodes, dtype=int)
    repeated_nodes[0:2] = [0, 1]
    cur_rep = 2

    # Preallocated edges array with 2 for u, v and n_steps for the number of edges
    edges = np.zeros((n_steps, 2), dtype=int)

    # One key optim. is that we can perform uniform sampl. from the repeated nodes
    # than degree proportional sampling, so we don't need to recompute degrees weig
    for step in range(n_steps):
        if np.random.rand() < alpha:
            u = repeated_nodes[np.random.randint(cur_rep)]
        else:
            u = np.random.randint(step +2)
        v = step + 2
        edges[step] = [u, v]
        degrees[u] += 1
        degrees[v] = 1

        # add nodes to repeated nodes list and increment position
        repeated_nodes[cur_rep:cur_rep+2] = [u, v]
        cur_rep += 2

    G = nx.Graph()
    G.add_edges_from(edges)
    return G
```

Part B

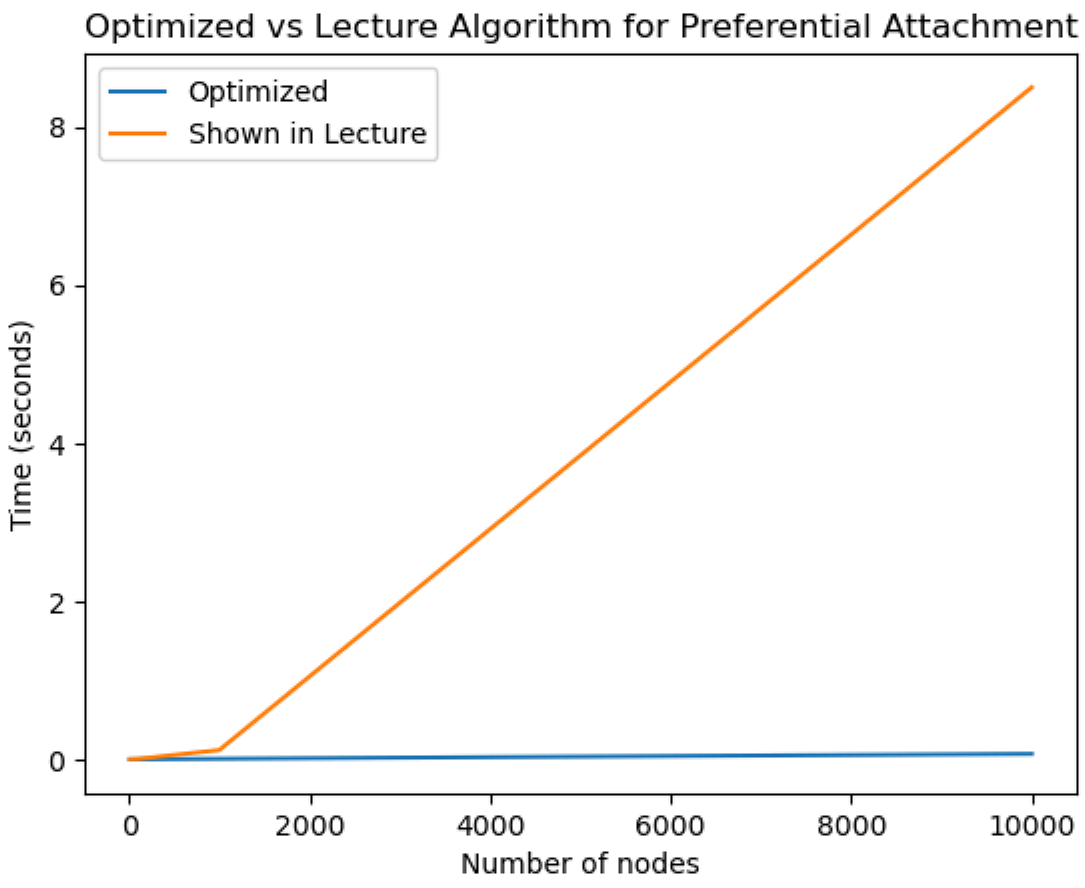
In [64]:
```
# CODE SOURCED FROM LECTURE NOTES AND ADAPTED TO FUNCTION: https://network-science-
def SlowPreferentialAttachment(alpha, n_steps):
    G = nx.Graph()
    G.add_edge(0, 1)

    # main loop
    for _ in range(n_steps):
        degrees = nx.degree(G)

        # determine u using one of two mechanisms
        if np.random.rand() < alpha:
            deg_seq = np.array([deg[1] for deg in degrees])
            degree_weights = deg_seq / deg_seq.sum()
            u = np.random.choice(np.arange(len(degrees)), p = degree_weights)
        else:
            u = np.random.choice(np.arange(len(degrees)))

        # integer index of new node v
        v = len(degrees)

        # add new edge to graph
        G.add_edge(u, v)
    return G
```

In [65]:

```
alpha = 4/5
hs = [1, 2, 3, 4]
ns = [10 ** h for h in hs]
optimized_times = []
slow_times = []

for n in ns:
    start = time.time()
    OptimizedPreferentialAttachment(n_steps=n, alpha=alpha)
    end = time.time()
    optimized_times.append(end - start)

    start = time.time()
    SlowPreferentialAttachment(alpha=alpha, n_steps=n)
    end = time.time()
    slow_times.append(end - start)

plt.plot(ns, optimized_times, label='Optimized')
plt.plot(ns, slow_times, label='Shown in Lecture')
plt.xlabel('Number of nodes')
plt.ylabel('Time (seconds)')
plt.title('Optimized vs Lecture Algorithm for Preferential Attachment')
plt.legend()
```

Out[65]:

```
<matplotlib.legend.Legend at 0x1ed7268fc50>
```



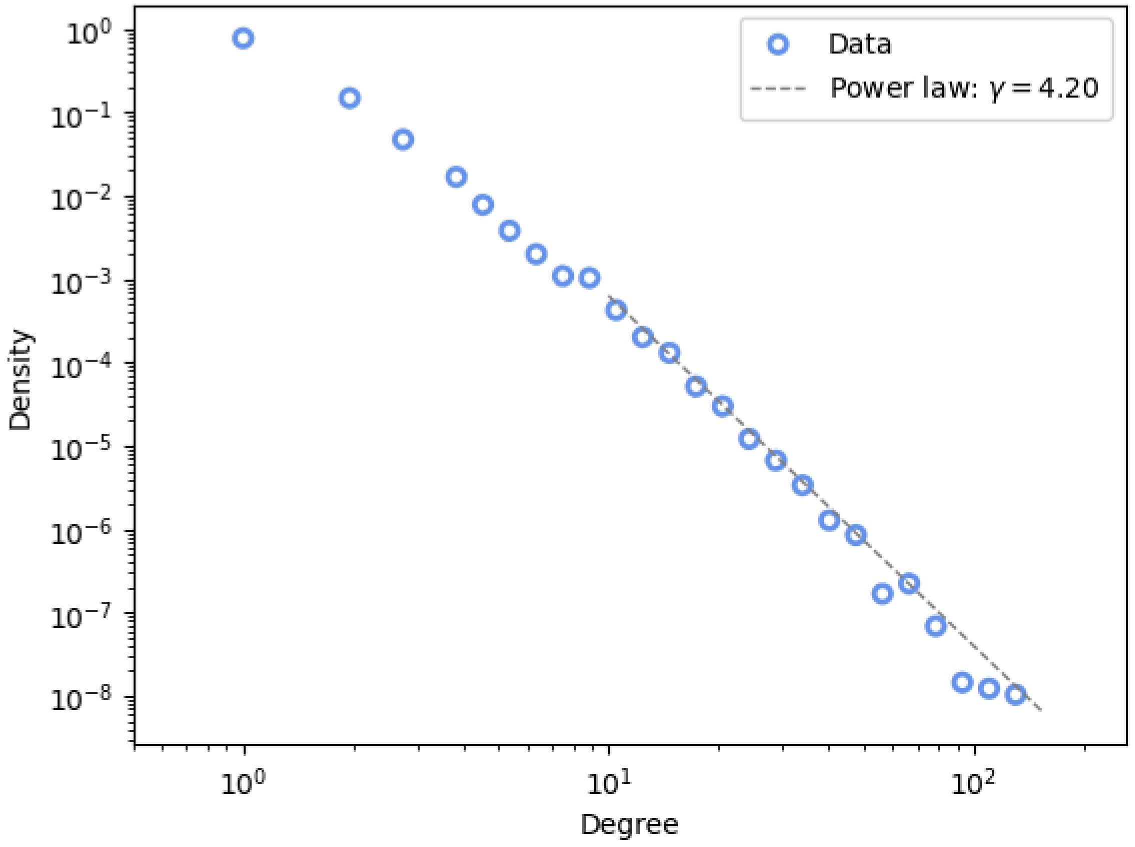
Part C

In [19]:
```python
# THIS CODE IS SOURCED FROM LECTURE NOTES: https://network-science-notes.github.io/
def degree_sequence(G):
    degrees = nx.degree(G)
    degree_sequence = np.array([deg[1] for deg in degrees])
    return degree_sequence

def log_binned_histogram(degree_sequence, interval = 5, num_bins = 20):
    hist, bins = np.histogram(degree_sequence, bins = min(int(len(degree_sequence)/
    bins = np.logspace(np.log10(bins[0]),np.log10(bins[-1]),len(bins))
    hist, bins = np.histogram(degree_sequence, bins = bins)
    binwidths = bins[1:] - bins[:-1]
    hist = hist / binwidths
    p = hist/hist.sum()

    return bins[:-1], p

def plot_degree_distribution(G, **kwargs):

    deg_seq = degree_sequence(G)
    x, p = log_binned_histogram(deg_seq, **kwargs)
    plt.scatter(x, p,  facecolors='none', edgecolors =  'cornflowerblue', linewidth
    plt.gca().set(xlabel = "Degree", xlim = (0.5, x.max()*2))
    plt.gca().set(ylabel = "Density")
    plt.gca().loglog()
    plt.legend()
    return plt.gca()
```

In [23]:
```
n = 10 ** 6
alphas = [0.5, 0.75, 1.0]
G0 = OptimizedPreferentialAttachment(n_steps=n, alpha=alphas[0])
G1 = OptimizedPreferentialAttachment(n_steps=n, alpha=alphas[1])
G2 = OptimizedPreferentialAttachment(n_steps=n, alpha=alphas[2])
GS = [G0, G1, G2]
```

In [40]:
```
i = 0
deg_seq = degree_sequence(GS[i])
cutoff  = 10
d_      = np.arange(cutoff, deg_seq.max(), 1)
gamma   = (1.6 + alphas[i]) / alphas[i]
power_law = 10*d_**(-gamma)

ax = plot_degree_distribution(GS[i], interval = 2, num_bins = 30)
ax.plot(d_, power_law,  linewidth = 1, label = fr"Power law: $\gamma = {gamma:.2f}$
ax.legend()
```

Out[40]:
```
<matplotlib.legend.Legend at 0x1ed3cf45a90>
```

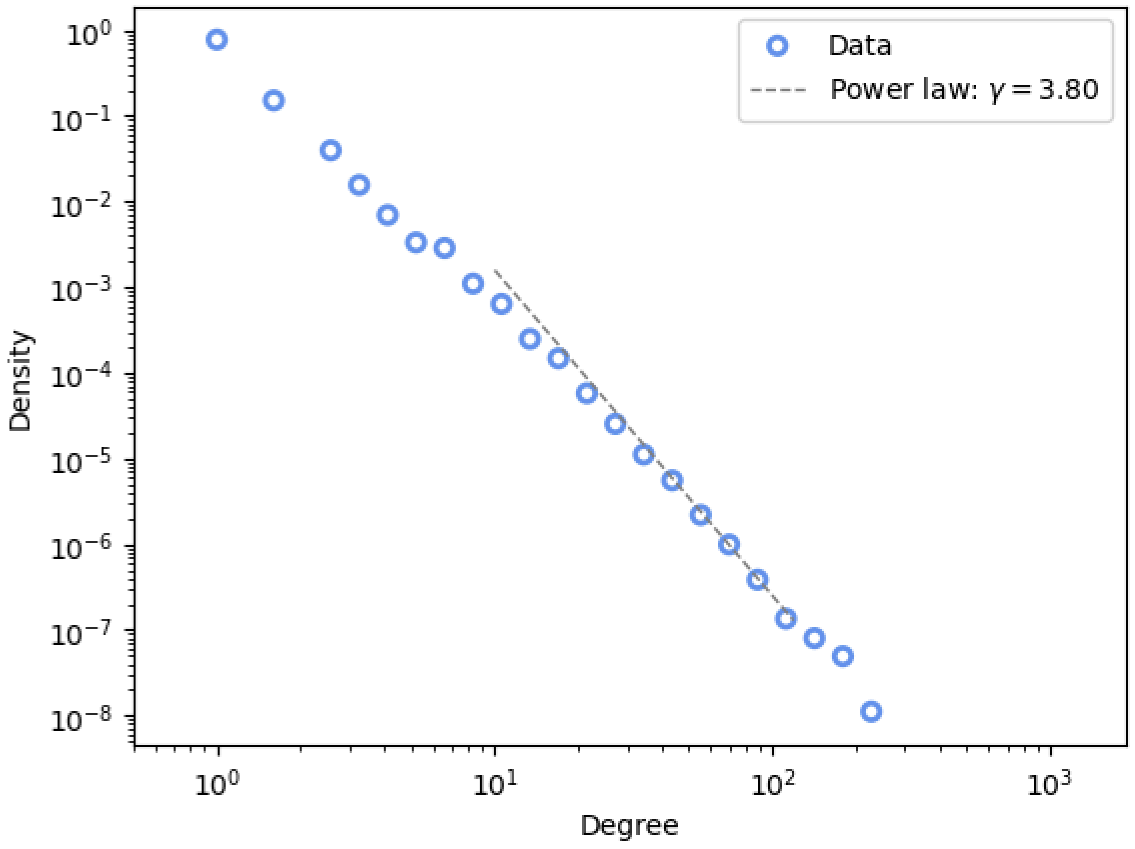


In [55]:
```
i = 1
deg_seq = degree_sequence(GS[i])
cutoff  = 10
d_      = np.arange(cutoff, 120, 1)
gamma   = (2.1 + alphas[i]) / alphas[i]
power_law = 10*d_**(-gamma)
```

```
ax = plot_degree_distribution(GS[i], interval = 2, num_bins = 30)
ax.plot(d_, power_law,  linewidth = 1, label = fr"Power law: $\gamma = {gamma:.2f}$
ax.legend()
```

Out[55]:

```
<matplotlib.legend.Legend at 0x1ed3d6fad50>
```

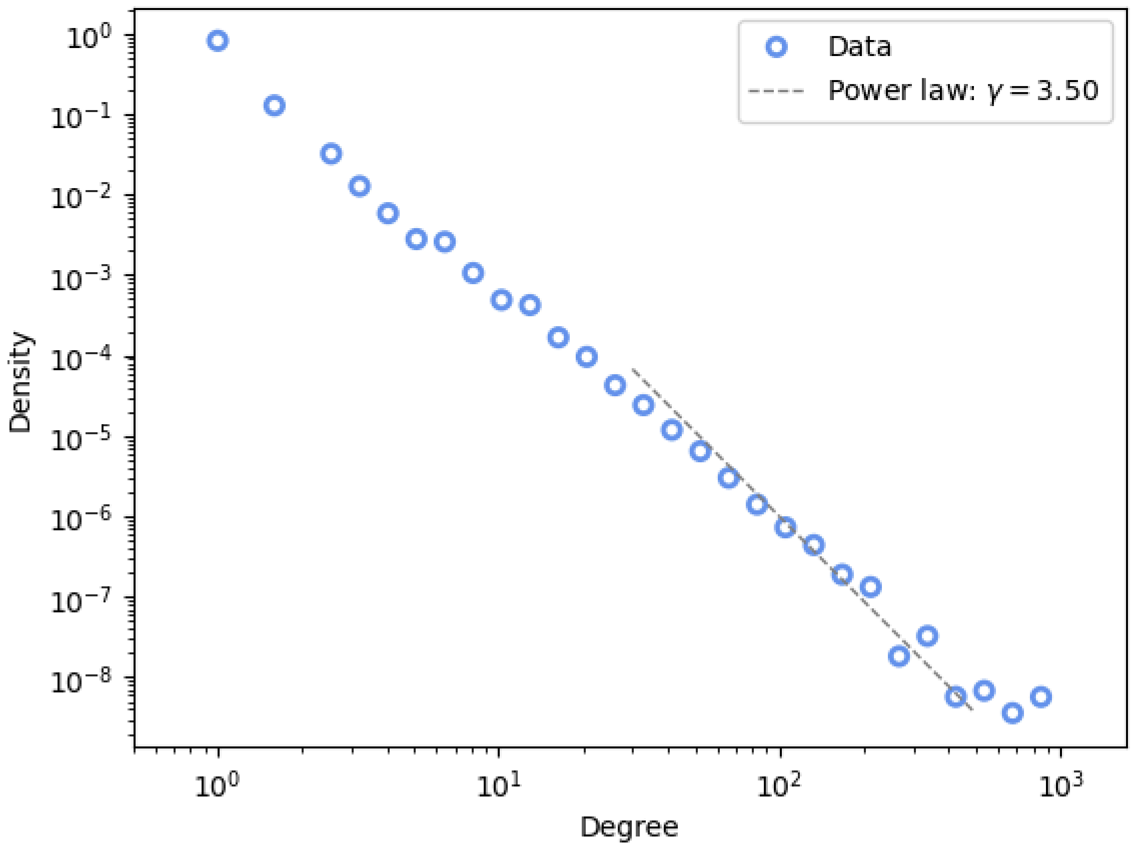


In [60]:

```
i = 2
deg_seq = degree_sequence(GS[i])
cutoff  = 30
d_      = np.arange(cutoff, 500, 1)
gamma   = (2.5 + alphas[i]) / alphas[i]
power_law = 10*d_**(-gamma)

ax = plot_degree_distribution(GS[i], interval = 2, num_bins = 30)
ax.plot(d_, power_law,  linewidth = 1, label = fr"Power law: $\gamma = {gamma:.2f}$
ax.legend()
```

Out[60]:

```
<matplotlib.legend.Legend at 0x1ed6db21310>
```


Data
Power law: $\gamma = 3.50$
Density
Degree
10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^0
10^1
10^2
10^3