```
!pip install powerlaw
```

In [1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
from collections import defaultdict
from statistics import mean
from scipy.sparse import dok_matrix, csr_matrix, triu, tril
from random import randint
import powerlaw as pl
import tail_estimation as te
from tqdm.notebook import tqdm
from utils import count_faster
```

Functions

In [2]:
```
################################################## Problem 2.1 ################
def distributionBin(x: 'input data',
                    B: 'number of bins'
                   ):

    """
    :param x: (list) list of real-valued integers interpreted as the input dat
    :param B: (positive int) number of log-scaled bins to create a histogram f

    distributionBin: Tuple, int -> Tuple, Tuple[Float]

    distributionBin takes a list of input data :x: assumed to be sampled from
    sized bins in such a way that the output function *integrates* to 1. The o
    estimated distribution values Y.
    """

    xmin = min(x)
    xmax = max(x)

    # creating the B+1 bin edges
    bin_edges = np.logspace(np.log10(xmin), np.log10(xmax), num=B+1)

    # using numpy's histogram function get distributions
    density, _ = np.histogram(x, bins=bin_edges, density=True)

    # obtaining bin midpoints for cleaner absciss
    log_be = np.log10(bin_edges)
    xout = 10**((log_be[1:] + log_be[:-1])/2)

    return xout, density
```

In [3]:
```
################################################### Problem 2.2 ################
def functionBin(x: 'preimage',
                y: 'function output for some function y | y_i=y(x)',
                B: 'number of bins'
               ):

    """
    :param x: (list) list of real-valued numbers interpreted as the preimage o
    :param y: (list) list of real-valued numbers interpreted as the values of 
    :param B: (positive int) number of log-scaled bins to create a histogram

    functionBin: Tuple, Tuple, int -> Tuple, Tuple

    functionBin takes an ordered list of sampled input values :x: and their co
    i.e. y(x)) and bins the input values into :B: log sized bins, averaging th
    the bin-midpoints X and the binned outputs.

    """

    xmin = min(x)
    xmax = max(x)

    # creating the B+1 bin edges
    bin_edges = np.logspace(np.log10(xmin), np.log10(xmax), num=B+1)

    # obtaining bin midpoints for cleaner absciss
    log_be = np.log10(bin_edges)
    bm = 10**((log_be[1:] + log_be[:-1])/2)

    # creating (input, output) pairs and sorting by input value
    fpairs = list(zip(x,y))
    fpairs.sort(key=lambda x: x[0])

    # creating (label, boundary) pairs using midpoint and right boundary for e
    mid_redge_pairs = list(zip(bm, bin_edges[1:]))

    # dictionary of values where key corresponds to the bin and values are tho
    bin_out_list = defaultdict(list)

    idx = 0
    for mid, redge in tqdm(mid_redge_pairs, desc='Binning Function'):
        while (fpairs[idx][0] < redge) & (idx < len(fpairs)-1):
            bin_out_list[mid].append(fpairs[idx])
            idx += 1

    # adding the last value
    bin_out_list[list(bin_out_list.keys())[-1]].append(fpairs[-1])

    xout = list(bin_out_list.keys())

    # y value is the average of each bin
    yout = [mean([i[1] for i in b]) for b in bin_out_list.values()]

    return xout, yout
```

In [4]:
```
############################################### Problem 2.3 ################
def simpleSF(n: 'graph size' = 10**4,
             s: 'number of samples' = 10,
             sparse: 'choice of numpy vs. scipy sparse matrix' = False
            ):

    """
    :param n: (positive int) total number of nodes in the desired graph after
    :param s: (positive int) total number of graphs created
    :param sparse: (boolean) False maps output to dense numpy ndarrays, True m

    simpleSF: Int, Int, Bool -> Tuple[np.ndarray OR scipy.sparse.csr_matrix]

    simpleSF takes a value for the size of the graph :n: and the number of sam
    size :n: by randomly selecting an edge E := (i,j) and attaching new incomi
    :s: (:n: x :n:) matrices which act as the adjacency matrix of each sample
    larger graph sizes.
    """

    outlist = list()
    pbar = tqdm(range(s), desc=f"Generating Samples")

    for i in pbar:

        # edgelist
        edgelist = {(0,1)}

        # selecting a random edge and making the incident nodes the fruits of
        # process repeats until t == n
        t = 2
        while t < n:
            m = randint(0, len(edgelist)-1)
            i,j = list(edgelist)[m]
            edgelist = edgelist | {(i, t), (j, t)}
            t += 1

        # mapping corresponding matrix values for all pairs in edgelist from 0
        if sparse:
            A = dok_matrix((n, n), dtype=int)
            for e in edgelist:
                A[e] = 1

            # currently only have upper triangle, so making the matrix symmetr
            # triu and tril authomatically change it from dok_matrix to csr_ma
            A = triu(A) + tril(A.T)

        else:
            A = np.zeros((n,n))
            for e in edgelist:
                A[e] = 1

            # currently only have upper triangle, so making the matrix symmetr
            A = np.triu(A) + np.tril(A.T)

        outlist.append(A)
```

```
    return outlist
```

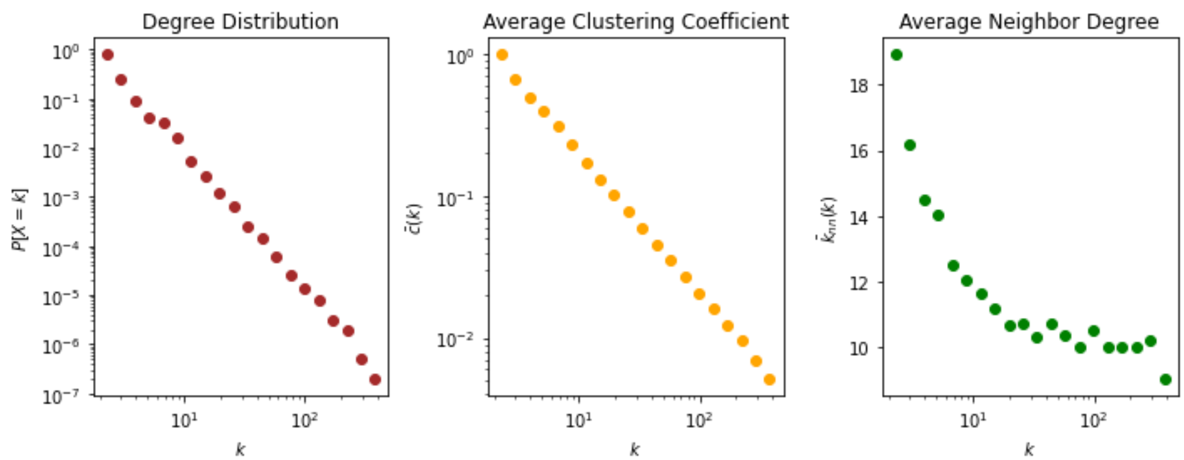
Tests

In [5]:
```
import networkx as nx
import itertools
```

In [6]:
```
################################################# Problem 2.4 ################

# creating graph ensemble G
G = simpleSF(sparse=True)

# setting number of bins
num_bins = 20

# preparing the plots

fig, axs = plt.subplots(1, 3)

################ plotting degree distribution ####################
degrees = list()
for g in G:
    # compressing ensemble by concatenating all degrees of all graphs into one
    degrees.extend(g.sum(axis=0).tolist()[0])


# binning into :num_bins: bins using function from 2.1
dist_x, dist_y = distributionBin(degrees, num_bins)

# plotting
axs[0].loglog(dist_x, dist_y, 'o', color='brown')
axs[0].set_title('Degree Distribution')
axs[0].set_ylabel(r'$P[X=k]$')
axs[0].set_xlabel(r'$k$')

################ plotting avg. clustering ####################

# using networkx to get average clustering and average neighbor degree
indexed_node_degrees = list()
indexed_clustering = list()
indexed_average_degree = list()

for g in G:
    # this needs to be np.from_numpy_matrix if not sparse
    graph = nx.from_scipy_sparse_array(g)
    for node in graph.nodes:
        indexed_node_degrees.append(graph.degree(node))
        indexed_clustering.append(nx.clustering(graph, nodes=[node])[node])
        indexed_average_degree.append(nx.average_neighbor_degree(graph, nodes=

# Using the function from 2.2 to plot these values
c_x, c_y = functionBin(indexed_node_degrees, indexed_clustering, num_bins)
kk_x, kk_y = functionBin(indexed_node_degrees, indexed_average_degree, num_bin

# plotting clustering on a loglog scale
axs[1].loglog(c_x, c_y, 'o', color='orange')
axs[1].set_title('Average Clustering Coefficient')
axs[1].set_xlabel(r'$k$')
axs[1].set_ylabel(r'$\bar{c}(k)$')

# plotting average neighbor degree on a semi-log scale
axs[2].plot(kk_x, kk_y, 'o', color='green')
```

```python
axs[2].set_xscale('log')
axs[2].set_title('Average Neighbor Degree')
axs[2].set_xlabel(r'$k$')
axs[2].set_ylabel(r'$\bar{k}_{nn}(k)$')

fig.set_size_inches(10, 4, forward=True)
fig.tight_layout()

fig.savefig('HW1_fig1_pk_ck_knnk.svg')
```

```
Generating Samples:   0%|          | 0/10 [00:00<?, ?it/s]

Binning Function:   0%|          | 0/20 [00:00<?, ?it/s]

Binning Function:   0%|          | 0/20 [00:00<?, ?it/s]
```



In [7]:
```python
## Powerlaw package fit

pl_fit = pl.Fit(degrees)
gamma_pl = pl_fit.power_law.alpha
print(f"Fit after x = {pl_fit.power_law.xmin}")
print(f"Estimated gamma: {gamma_pl}")
```

```
Calculating best minimal value for power law fit
Fit after x = 16.0
Estimated gamma: 2.9707989546170106
```

In [8]:
```python
## Hill, Moments, and Kernel estimators

# exporting the data to a dat file
import struct

# sending degree counts to a .dat file that can be used with the package
deg_data = count_faster(degrees)
deg_data.sort(key=lambda x: x[0])
deg_data = np.array(deg_data, dtype=np.int64)
np.savetxt('synthetic_scale_free_degree.dat', deg_data, fmt='%i')
```

In [9]:
```
!python tail_estimation.py synthetic_scale_free_degree.dat HW1_Estimator_plots
```

```
========== Tail Index Estimation ==========
Number of data entries: 100000
========================
Selected AMSE border value: 1.0000
Selected fraction of order statistics boundary for AMSE minimization: 1.0000
========================
Adjusted Hill estimated gamma: 2.8598126560047312
**********
Moments estimated gamma: 2.8530630588253105
**********
Kernel-type estimated gamma: 2.8626349509553295
**********
Elapsed time (total): 51.62576174736023
```

In [10]:
```

hill_gamma = 2.8576419049285056
moments_gamma = 2.8602272493254426
kernel_gamma = 3.0387108557966616

# Plotting Distribution with all of its estimates
plt.loglog(dist_x, dist_y, 'o', color='brown')
plt.plot(dist_x, [k**(-1*pl_fit.power_law.alpha) for k in dist_x], linestyle='
         alpha=0.8)
plt.plot(dist_x, [k**(-1*hill_gamma) for k in dist_x], linestyle='dotted', col
plt.plot(dist_x, [k**(-1*moments_gamma) for k in dist_x], linestyle='dotted',
plt.plot(dist_x, [k**(-1*kernel_gamma) for k in dist_x], linestyle='dotted', c
plt.legend()
plt.title('Binned Degree Distribution with Estimates')
plt.ylabel(r'$P[X=k]$')
plt.xlabel(r'$k$')
plt.tight_layout()

plt.savefig('HW1_fig2_degree_dist_estimates.svg')
```