```python
from collections import defaultdict
import numpy as np
from statistics import mean
from scipy.sparse import dok_matrix, csr_matrix, triu, tril
from random import randint

from tqdm.notebook import tqdm

"""
utils.py

Written by Sagar Kumar, April 2023

This is a Python script with useful functions which I have written and which are utilized throughout the scripts which
have been written for Problem 3, Homework 2 in NETS6116.

Some come from HW1.
"""


def count_faster(l):
    dic = {}

    for i in l:

        if i in dic.keys():
            dic[i] += 1

        else:
            dic[i] = 1

    return [(key, dic[key]) for key in dic.keys()]



def distributionBin(x: 'input data',
                    B: 'number of bins'
                    ):

    """
    :param x: (list) list of real-valued integers interpreted as the input data sampled from a presumed heavy-tail distribution
    :param B: (positive int) number of log-scaled bins to create a histogram from x

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

    distributionBin takes a list of input data :x: assumed to be sampled from some heavy-tailed distribution and bins in :B: log
    sized bins in such a way that the output function *integrates* to 1. The output is a list of bin-midpoints X and the binned
    estimated distribution values Y.
    """

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

    # creating the B+1 bin edges
    bin_edges = np.logspace(max(1,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


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 of the function y(x)
    :param y: (list) list of real-valued numbers interpreted as the values of function y applied to each input value in x
    :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 corresponding outputs under some function :y: (
    i.e. y(x)) and bins the input values into :B: log sized bins, averaging the output values within each bin. The output is
    the bin-midpoints X and the binned outputs.

    """

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

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

    # obtaining bin midpoints for cleaner absciss
```

```python
    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 each bin
    mid_redge_pairs = list(zip(bm, bin_edges[1:]))

    # dictionary of values where key corresponds to the bin and values are those which fit in the bin
    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


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 completing its percolation
    :param s: (positive int) total number of graphs created
    :param sparse: (boolean) False maps output to dense numpy ndarrays, True maps them to scipy sparse csr_matrix

    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 samples :s: and percolates :s: scale-free graphs of
    size :n: by randomly selecting an edge E := (i,j) and attaching new incoming nodes to both i and j. The output is a list of
    :s: (:n: x :n:) matrices which act as the adjacency matrix of each sample graph. Choosing scipy sparse matrix allows for
    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 an incoming cherry to create a triangle
        # 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 -> 1
        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 symmetric about the main diagonal
            # triu and tril authomatically change it from dok_matrix to csr_matrix
            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 symmetric about the main diagonal
            A = np.triu(A) + np.tril(A.T)

        outlist.append(A)

    return outlist

```