Modelling Independent Cascade Model in a Undirected Scale Free Network

A scale free network has a few nodes that are highly connected to other nodes in the network. The presence of such nodes with a much higher degree than most other nodes will give the degree distribution a long tail. A scale-free network is one with a power-law degree distribution. For an undirected network.

$$P_{deg}(k) \propto k^{-\gamma}$$

One can recognize that a degree distribution has a power-law form by plotting it on a log-log scale. As shown in the above scatter plot, the points will tend to fall along a line.

For this investigation, we choose to use the Barabasi-Albert model. The preferential attachment algorithm is suh=ch that the probability that a new node connects to an existing node i is

$$\pi(k_i) = \frac{k_i}{\sum_j k_j}$$

For the BA model, the specific growth and attachment rule leads to $\gamma \approx 3$.

The γ value can be affected by factors like :

1. Preferential attachment mechanism
2. Growth and attachment rules
3. Initial degree of nodes (m)
4. Network size (secondary effect)

Several natural and human-made systems, including the internet, the world wide web, citation networks, and some social networks are thought to be approximately scale-free.

Information cascade is a phenomenon in which a number of people make the same decision in a sequantial information. We can generally accept this as a two-step process. First, an individual encounters a scenario with a decision (binary). Second, the decision is influenced by the individual observing others' choice and the apprarent outcomes. There are several information cascade models.

We choose to investigate the Independence Cascade Model(ICM) of Information Diffusion. Node can have three states :

- Inactive - node unaware of information orYb not influenced
- Active - Node already influenced by information in diffusion
- Actived - nodes activated other nodes at prev round, but can not activate other nodes anymore.

The process runs in discrete steps. At the beginning of ICM process, few nodes are given the information known as seed nodes. Upon receiving the information these nodes become active. In each discrete step, an active node tries to influence one of its inactive neighbors. In spite of its success, the same node will never get another chance to activate the same inactive neighbor. The success depends on the propagation probability of their tie. Propagation Probability of a tie is the probability by which one can influence the other node.

The process terminates when no further nodes became activated from inactive state.

We set the propagation probability to be

$$p_{ij} = \frac{q}{k_j}$$

Where j is the target node and i is the active node. q is the threshold, a hyper parameter.

The goal - Influence Maximization

Similar to marketing or broadcasting in the real world. We can choose better seeds that result in a higher number of ultimate active nodes.

Here are some selection policies we will consider:

i. Random ii. Degree iii. Fixed degree iv. Friend degree v. Fixed friend degree

Fixed degree : Using degree ranking has a problem. If we pick a node with high degree but a fraction of its neighbours are actived, these nodes will no longer be considered any more. So the effective degree of the node is actually lower. To fix this, update nodes' (effective) degree after each selection, i.e. after selecting one node as a seed, update the degree its neighbours with -1 to make sure those nodes don't consider this seed anymore.

This helps avoid local seeds gathering.

Neighbour degree : Select seed by nodes' centrality score.

$$C_i = k_i + \sum_j k_j A_{ij}$$

The question : Is it worth it ?

The reason we investigate the above is because for a particular network and q, does the seeding protocol matter? Transitioning from random to non-random requires knowing the network topology, and adding fixed detail to the seeding protocol requires extra computing work as well. All these extra work can only be justified if there's siginificant improvement in final influenced size.

We focus on a quantity known as E, the effective influence:

$$E = \frac{< D >}{p_{seed} * n} - 1$$

Where $< D >$ is the mean influenced nodes at termination, the -1 accounts for the seed itself. Think of this quantity as for each seed that I sowed, how many other nodes was it able to influence.

Bonus : Homophily - popular nodes befriending each other.

[Scale-free network] https://mathinsight.org/scale_free_network

[Building scale-free/Barabasi-Albert Network] https://github.com/AlxndrMlk/Barabasi-Albert_Network

[How to Code Independent Cascade Model of Information Diffusion] http://home.iitj.ac.in/~suman/articles/detail/how-to-code-independent-cascade-model-of-information-diffusion/#:~:text=Independent%20Cascade%20Model%20(ICM)%20is,by%20the%20informatior

[Python implementation of ICM & Selection Policies] https://github.com/cbhua/independent-cascade?tab=readme-ov-file

In [ ]:
```
import numpy as np
from matplotlib import pyplot as plt
from timeit import timeit
import networkx as nx
```

0 : Implementing Scale-free Network & ICM Simulation

Keep these constant :

- Number of nodes n = 10,000.
- m = 10.
- Set initial seeds be $p_{seed} = 0.01$

In [ ]:
```
class Network():
    def __init__(self,num_nodes):
        self.adj = {i:set() for i in range(num_nodes)}
        self.num_edge = 0
        self.num_nodes=num_nodes

    def add_edge(self,i,j):
        self.adj[i].add(j)
```

```
        self.adj[j].add(i)
        self.num_edge+=1

    def neighbors (self,i):
        return self.adj[i]

    def edge_list(self):
        return [(i,j) for i in self.adj for j in self.adj[i] if i<j]\

    def degree_distribution(self) -> list:
        '''Returns degree of each node'''
        return [len(self.adj[i]) for i in range(self.num_nodes)]
```

In [ ]:

```
class Barabasi_Albert_Network(Network):
    """Returns a random graph according to the Barabási–Albert preferential
    Attachment model.

    A graph of ``n`` nodes is grown by attaching new nodes each with ``m``
    Edges that are preferentially attached to existing nodes with high degree.

    Parameters
    ----------
    n : int
        Number of nodes
    m : int
        Number of edges to attach from a new node to existing nodes
    seed : int, optional
        Seed for random number generator (default=None).

    Returns
    -------
    G : Graph

    Raises
    ------
    ValueError
        If ``m`` does not satisfy ``1 <= m < n``. """

    def __init__(self, num_nodes, m):
        super().__init__(num_nodes)
        self.num_nodes = num_nodes
        self.m = m

        if m<1 or m>=num_nodes:
            raise ValueError('Barabasi-Albert network must have 1<=m<n')

        # Target nodes for new edges
        targets= list(range(m))
        # List of existing nodes, with nodes repeated one for each half edge
        repeated_nodes=[]
        # Start adding the other n-m nodes. The first node is m.
        for node in range(self.num_nodes-m):
            for i,j in zip([node]*m,targets):
                self.add_edge(i,j)
            repeated_nodes.extend(targets)
            repeated_nodes.extend([node]*m)
```

```
            # Choose m unique nodes from the existing nodes
            # Pick uniformly from repeated_nodes (preferential attachment)
            targets = np.random.choice(repeated_nodes,m,replace=False)
```

In [ ]:
```
class ICM_Model():
    """ I: Inactive, A:Active, D:Actived"""
    def __init__(self, network : nx.Graph, q, initial_seeds = None , p_seed=0.05) -
        self.network = network
        self.q = q
        self.p_seed = p_seed

        # I,A,D states
        self.I = {n for n in self.network}
        self.A = set()
        self.D = set()

        # Initially activate a small fraction of the population
        if not initial_seeds:
            initial_seeds = np.random.choice(list(self.I), size=int(self.p_seed*sel

        self.A.update(initial_seeds)
        self.I.difference_update(self.A)

    def run(self):
        '''Runs simulation for a cycle'''

        next_A = set()
        for node in self.A:
            for j in self.network.neighbors(node):
                if j in self.I and np.random.rand() < (self.q/self.network.degree(j
                    next_A.add(j)

        self.D.update(self.A)
        self.A = next_A
        self.I.difference_update(next_A)

    def run_to_extinction(self) -> tuple[list[int],list[int],list[int]]:
        '''Run simulation until no more active node is left, then returns time seri

        I_list, A_list, D_list = [len(self.I)],[len(self.A)],[len(self.D)]

        while self.A:
            self.run()

            I_list.append(len(self.I))
            A_list.append(len(self.A))
            D_list.append(len(self.D))

        return I_list, A_list, D_list
```

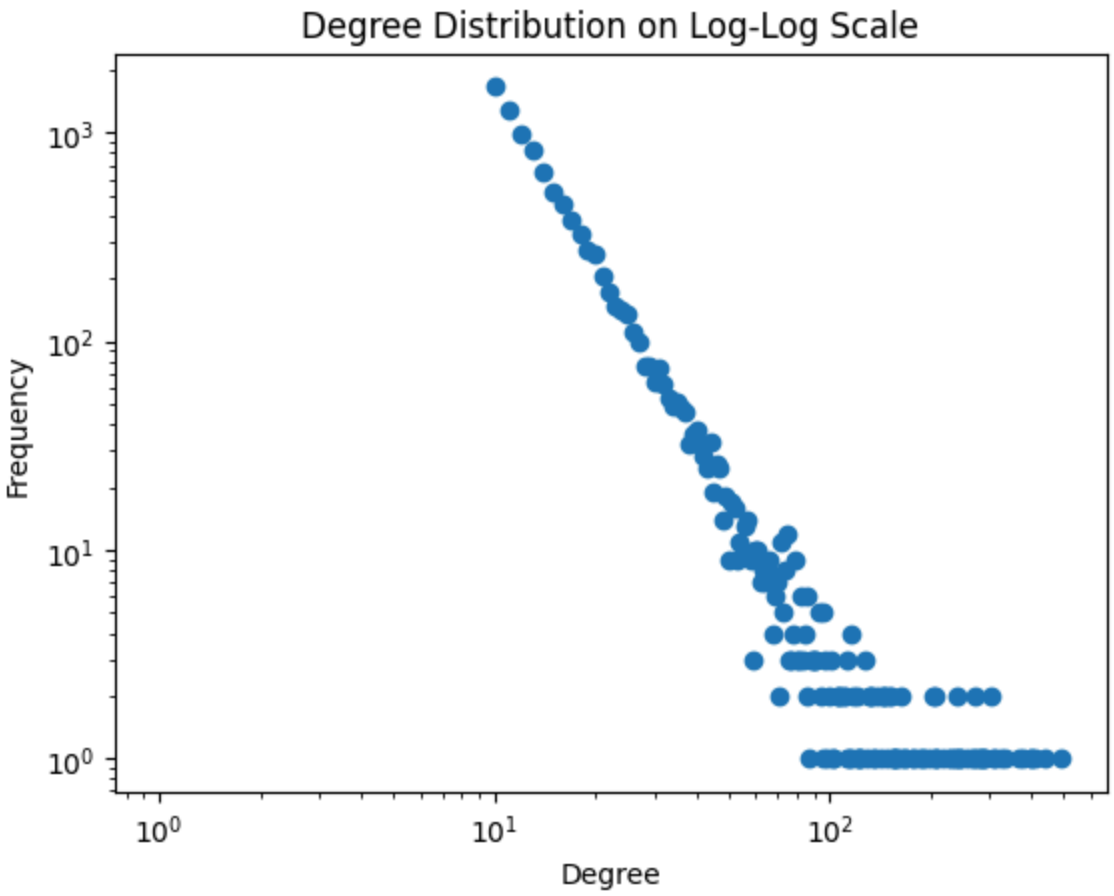
In [ ]:
```
SETUP_CODE="""from __main__ import Network
from __main__ import Barabasi_Albert_Network
import networkx as nx
n=10000
m=10"""
```

```
t_scratch = timeit(stmt='Barabasi_Albert_Network(n,m)',setup=SETUP_CODE,number=1)
t_nx = timeit(stmt='nx.barabasi_albert_graph(n,m)', setup=SETUP_CODE,number=1)

print('Time taken to build BA network from scratch = {}'.format(t_scratch))
print('Time taken to build BA network with networkx = {}'.format(t_nx))
```

```
Time taken to build BA network from scratch = 105.66762860000017
Time taken to build BA network with networkx = 0.3251053999993019
```

In [ ]:

```
n = 10000
m = 10
G = nx.barabasi_albert_graph(n,m)

# Plot degree distribution on a log-log scale
degree_sequence = [d for n, d in G.degree()]
degree_count = np.bincount(degree_sequence)
deg = np.arange(len(degree_count))

plt.scatter(deg,degree_count)
plt.yscale('log')
plt.xscale('log')
plt.title("Degree Distribution on Log-Log Scale")
plt.xlabel("Degree")
plt.ylabel("Frequency")
plt.show()
```



We will use networkx instead because the graph generation is much faster.

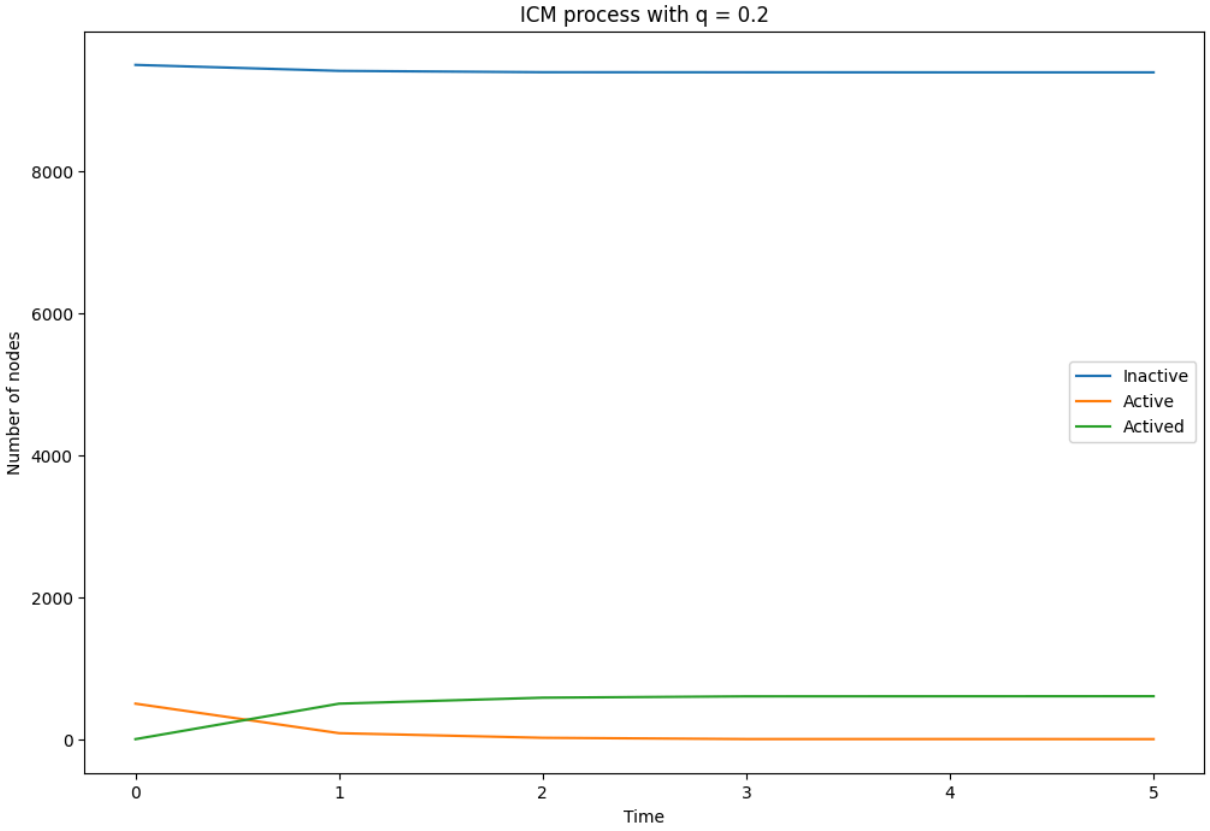
1 : States in ICM over time.

Inactive - I Active - A Actived - D

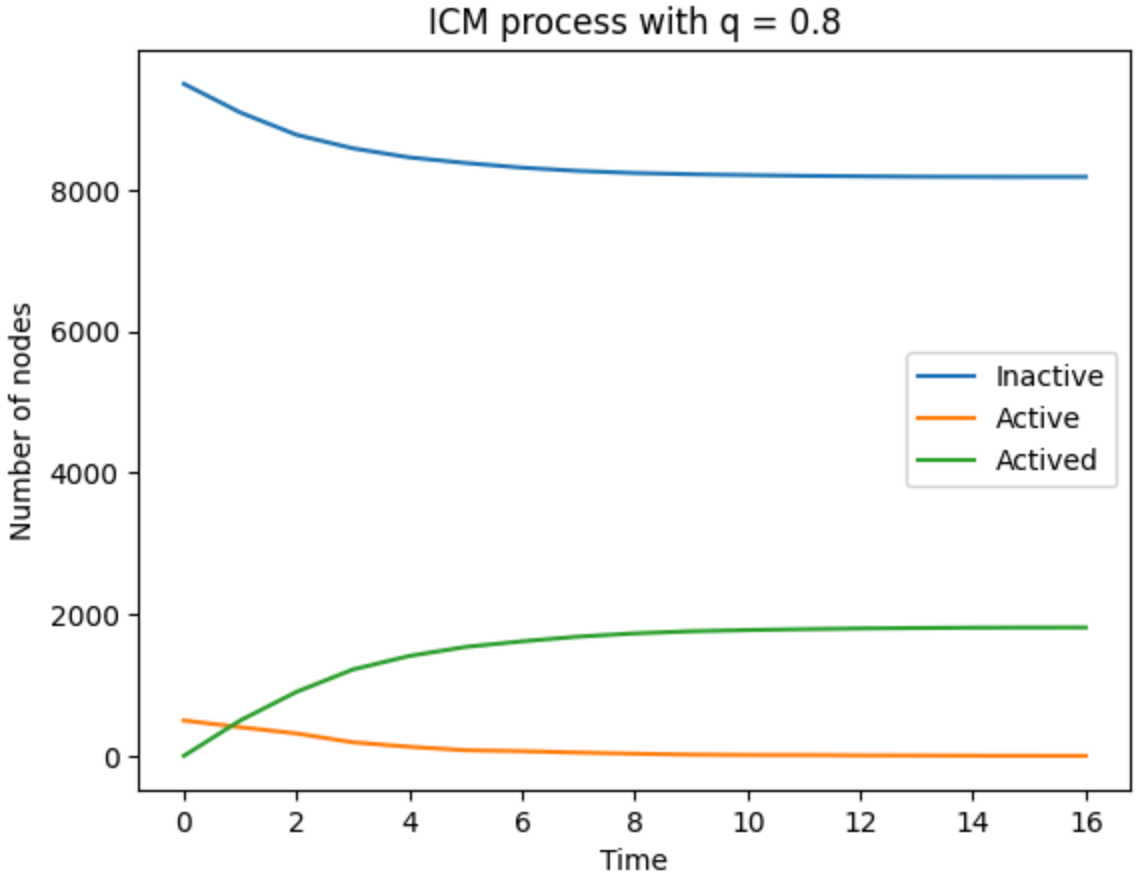
Simulate the ICM process. Run the process until extinction - we only have inactive and actived nodes. For a few q values. Plot time series for I,A,D.

Randomly pick the seeds.

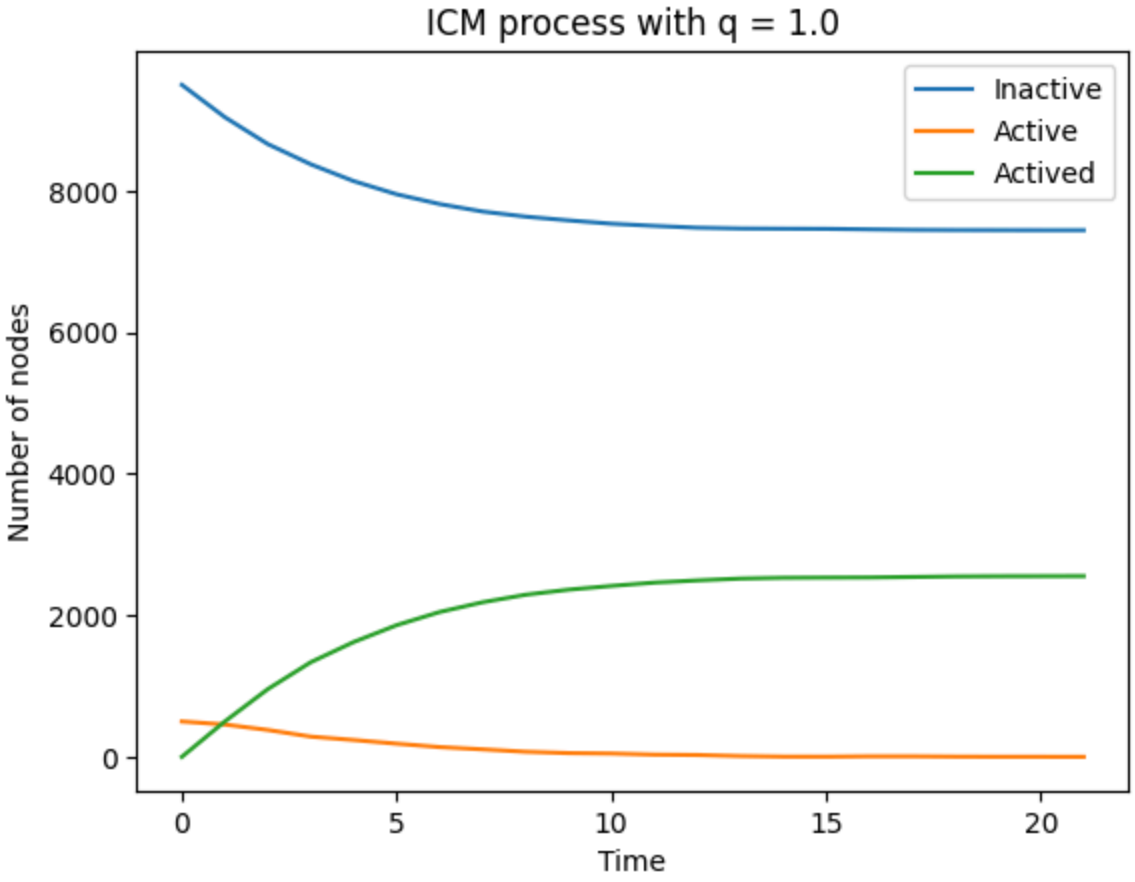
In [ ]:
```
q_array = [0.2,0.8,1.0]
p_seed = 0.05

plt.figure(figsize=(12,8))
for q in q_array:
    simulation = ICM_Model(G,q,p_seed=p_seed)
    I,A,D = simulation.run_to_extinction()

    plt.plot(I, label='Inactive')
    plt.plot(A, label='Active')
    plt.plot(D, label='Actived')
    plt.xlabel('Time')
    plt.ylabel('Number of nodes')
    plt.title('ICM process with q = {}'.format(q))
    plt.legend()
    plt.show()


    # Ratio of influenced size/seeds
    print('Effective influence = {}'.format((D[-1]/(n*p_seed))-1))
```




```
Effective influence = 0.20999999999999996
```




```
Effective influence = 2.63
```




```
Effective influence = 4.11
```

2 : Investigate influenced size versus propagation probability threshold by random seeding

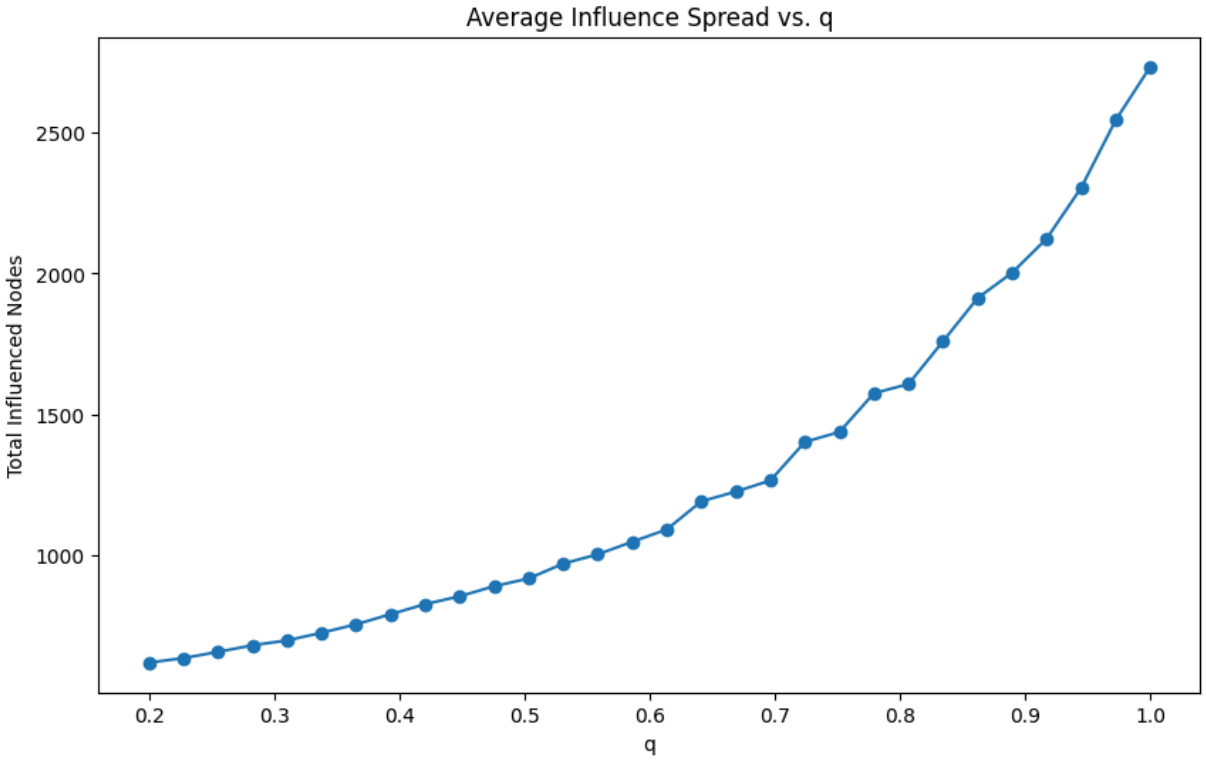
For a range of q values. Investigate the final size of influenced node, D , in the population, ideally averaged over a few network.

Are there any correlation ? Is there a 'threshold' for q where the dynamic of information cascading changes ?

In [ ]:
```python
def average_D_vs_q(G, q_array, initial_seeds=None, p_seed=0.05, runs=30):
    avg_D = []

    for q in q_array:
        D_values = []
        for _ in range(runs):
            simulation = ICM_Model(G,q,initial_seeds,p_seed)
            I,A,D = simulation.run_to_extinction()
            D_values.append(D[-1])
        avg_D.append(np.mean(D_values))

    return avg_D
```

In [ ]:
```
p_seed=0.05
q_array = np.linspace(0.2,1.0,30)
avg_D = average_D_vs_q(G, q_array,p_seed=p_seed)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(q_array,avg_D, marker='o')
plt.xlabel('q')
plt.ylabel('Total Influenced Nodes')
plt.title('Average Influence Spread vs. q')
plt.show()
```



No change in influence spread dynamic as a function of q.

3 : Probability that a node is never influenced

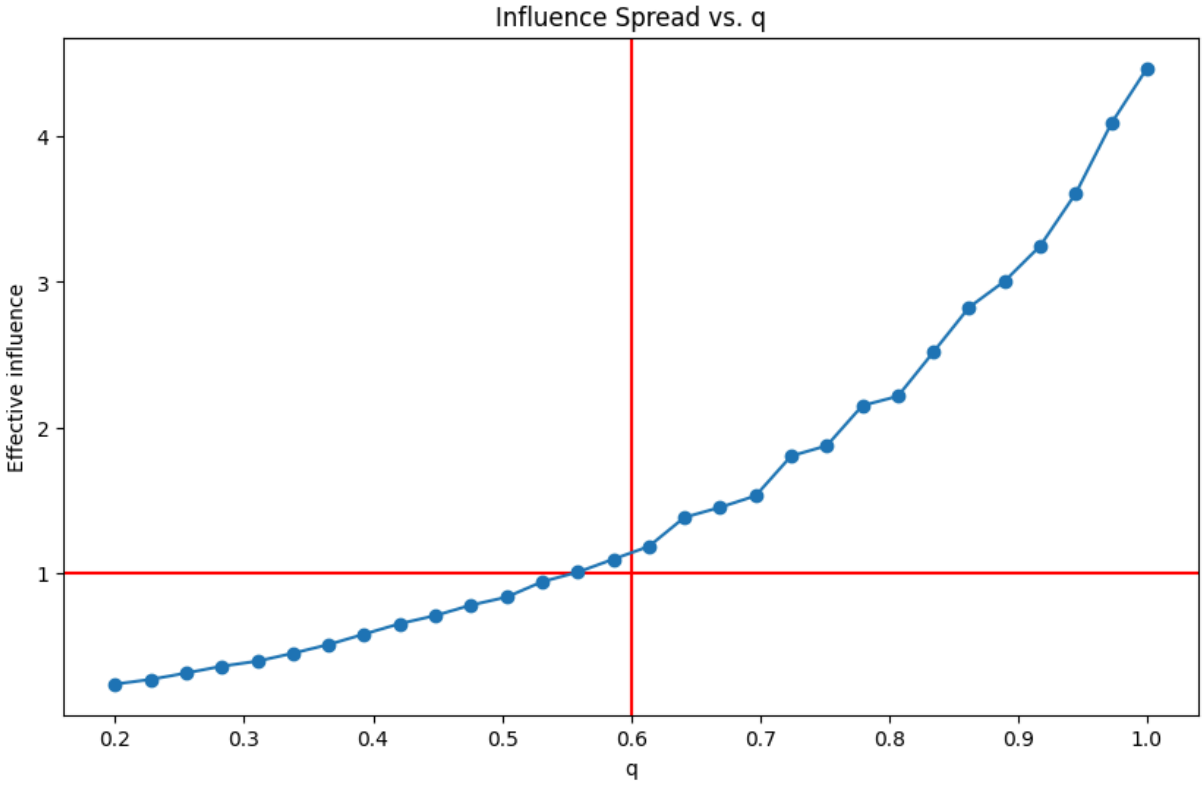
The suitable q value for this section will come from Q2.

Find a way to find analytical expression of probability a node is never influenced, s_i. Compute the vector for it. Then find the mean.

Use the above to estimate 'effectiveness' - ratio of final influence proportion to seed proportion.Final influence proportion is probably $1- < s >$.

In [ ]:
```
# TODO : Derive analytical expression then plot against graphs in 2

# Simulated influence effectiveness vs.
```

```
plt.figure(figsize=(10,6))
plt.axvline(0.6,color='r')
plt.axhline(1.0,color='r')
plt.plot(q_array,np.array(avg_D)/(p_seed*n)-1, marker='o',label='Simulation')
plt.xlabel('q')
plt.ylabel('Effective influence')
plt.title('Influence Spread vs. q')
plt.show()
```



The analytical derivations are presented at the bottom.

4 : Explore degree seeding protocol

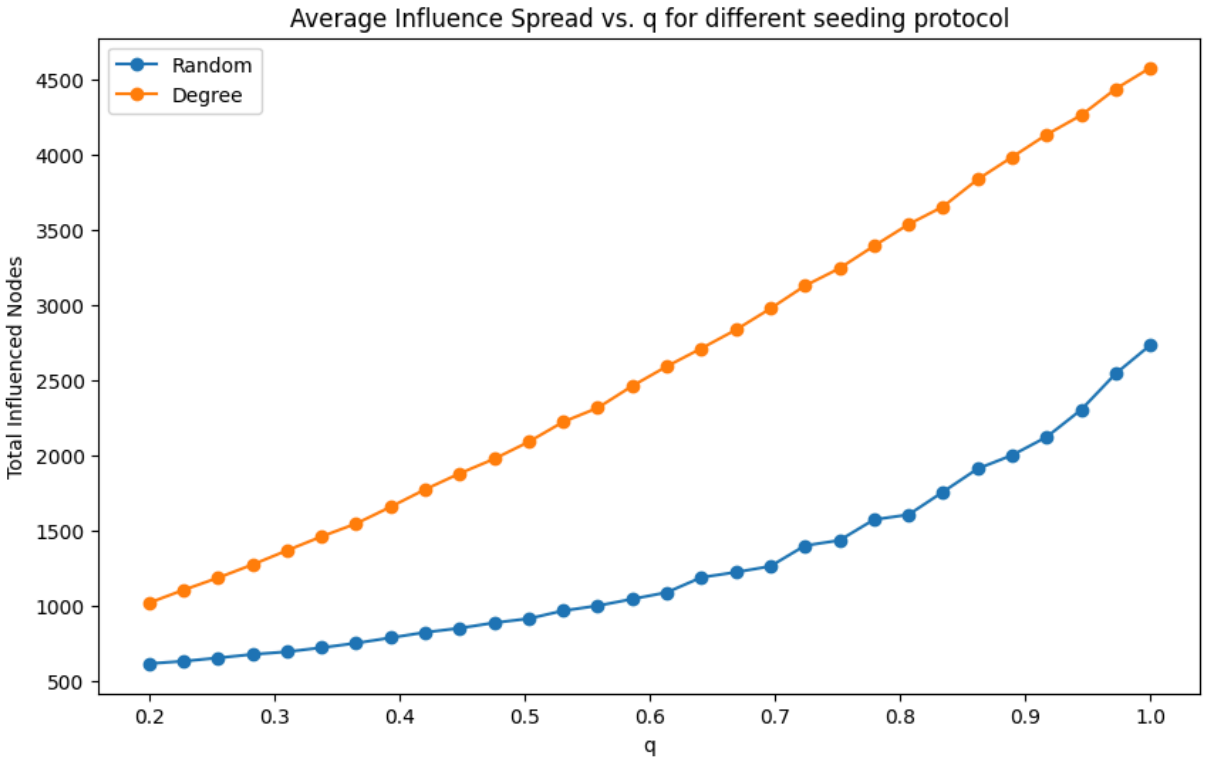
Now, instead of randomly picking seeds. Compute the degree vector and pick those with highest degrees. Repeat above analysis of influenced size as a function of q.

In [ ]:
```
def degree_seeding(G, num_seeds):

    nodes_degree_sorted = sorted(G.degree(), key=lambda x:x[1], reverse=True)
    return [n for n,d in nodes_degree_sorted[:num_seeds]]
```

In [ ]:
```
deg_seeds = degree_seeding(G, int(n*p_seed))
avg_D_deg = average_D_vs_q(G, q_array, deg_seeds)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(q_array,avg_D, marker='o', label='Random')
plt.plot(q_array,avg_D_deg, marker='o', label='Degree')
plt.legend()
```

```
plt.xlabel('q')
plt.ylabel('Total Influenced Nodes')
plt.title('Average Influence Spread vs. q for different seeding protocol')
plt.show()
```



5 : Explore friend degree seeding protocol

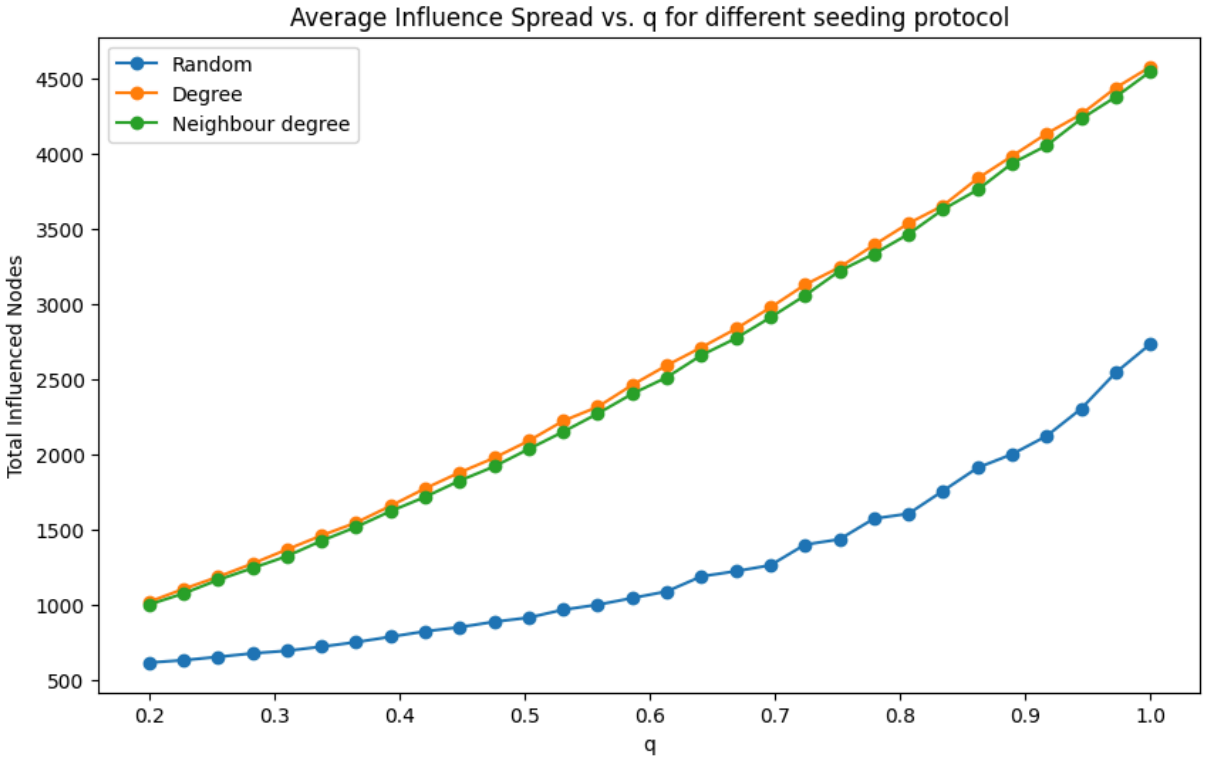
Compute centrality score c_i of a node by adding up their own degree and their neighbours degree. Pick nodes of highest centrality score as seed. Repeat analyis.

In [ ]:
```
def neighbour_degree_seeding(G, num_seeds):
    centrality = {}
    for node in G:
        centrality[node] = G.degree[node]
        for j in G.neighbors(node):
            centrality[node] += G.degree[j]

    nodes_centrality_sorted = sorted(centrality.items(), key=lambda x:x[1], reverse
    return [n for n,d in nodes_centrality_sorted[:num_seeds]]
```

In [ ]:
```
neigh_deg_seeds = neighbour_degree_seeding(G, int(n*p_seed))
avg_D_neigh_deg = average_D_vs_q(G, q_array, neigh_deg_seeds)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(q_array,avg_D, marker='o', label='Random')
plt.plot(q_array,avg_D_deg, marker='o', label='Degree')
plt.plot(q_array, avg_D_neigh_deg, marker='o', label='Neighbour degree')
plt.legend()
plt.xlabel('q')
```

```
plt.ylabel('Total Influenced Nodes')
plt.title('Average Influence Spread vs. q for different seeding protocol')
plt.show()
```



6 : Explore fixed degree seeding protocol

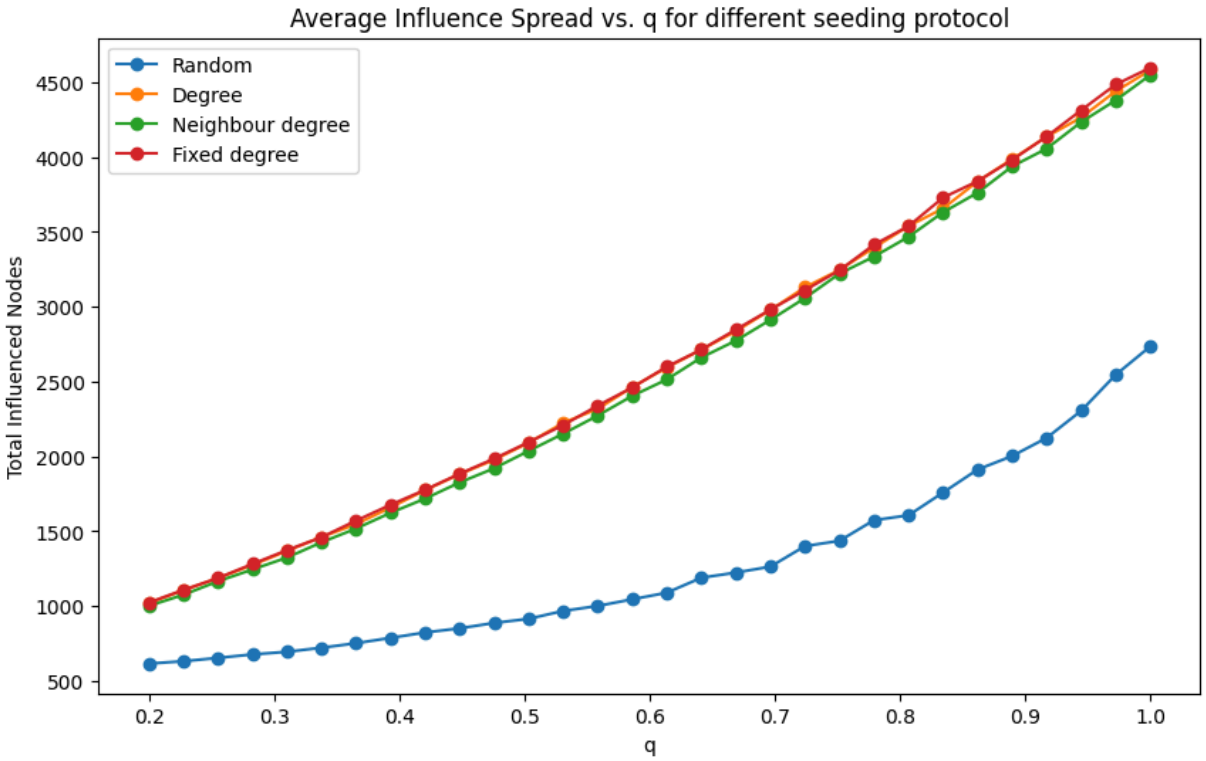
Start with a seed, then for all the neighbours of the seed, -1 with their degree. Pick node of next highest degree as next seed. Repeat degree adjustmenet and picking until $p_{seed} * n$ nodes selected. Repeat analysis.

Compare time complexity for the process so far.

In [ ]:
```
def fixed_degree_seeding(G, num_seeds):

    seeds = []
    degree = list(G.degree())

    for _ in range(num_seeds):
         nodes_degree_sorted = sorted(degree, key=lambda x:x[1], reverse=True)
         curr = nodes_degree_sorted[0][0]
         seeds.append(curr)
         degree[curr] = (curr,-1)
         for j in G.neighbors(curr):
              degree[j]= (j, degree[j][1]-1)

    return seeds
```

In [ ]:
```
fixed_deg_seeds = fixed_degree_seeding(G, int(n*p_seed))
avg_D_fixed_deg = average_D_vs_q(G, q_array, fixed_deg_seeds)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(q_array,avg_D, marker='o', label='Random')
plt.plot(q_array,avg_D_deg, marker='o', label='Degree')
plt.plot(q_array, avg_D_neigh_deg, marker='o', label='Neighbour degree')
plt.plot(q_array,avg_D_fixed_deg, marker='o', label='Fixed degree')
plt.legend()
plt.xlabel('q')
plt.ylabel('Total Influenced Nodes')
plt.title('Average Influence Spread vs. q for different seeding protocol')
plt.show()
```



7 : Explore fixed friend degree seeding protocol

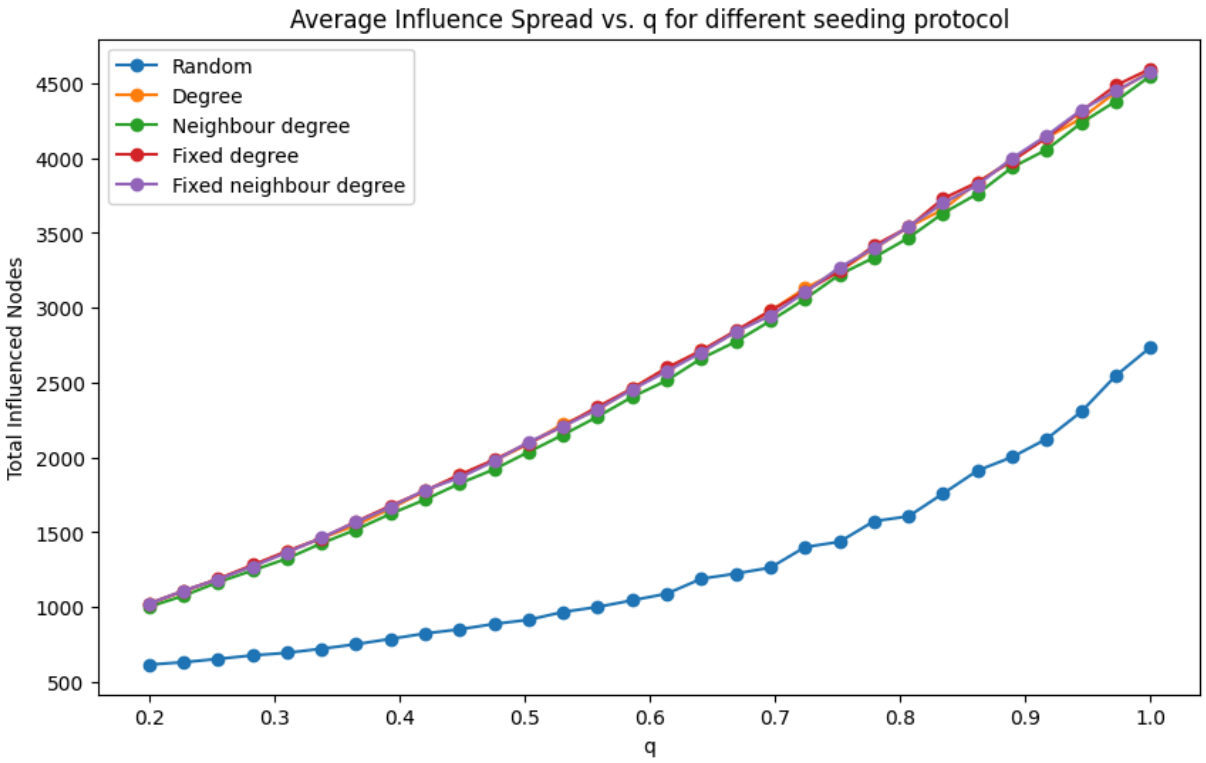
Self-explanatory. If fixed seeding protocol takes too long just ditch and do degree vs friend degree investigation.

In [ ]:
```
def fixed_neighbour_degree_seeding(G, num_seeds):

    seeds = []
    centrality = {}
    for node in G:
        centrality[node] = G.degree[node]
        for j in G.neighbors(node):
            centrality[node] += G.degree[j]
```

```
    for _ in range(num_seeds):
        nodes_centrality_sorted = sorted(centrality.items(), key=lambda x:x[1], re
        curr = nodes_centrality_sorted[0][0]
        seeds.append(curr)
        centrality[curr]=-1
        for j in G.neighbors(curr):
            centrality[j]-=G.degree[curr]

    return seeds
```

In [ ]:
```
fixed_neigh_deg_seeds = fixed_neighbour_degree_seeding(G, int(n*p_seed))
avg_D_fixed_neigh_deg = average_D_vs_q(G, q_array, fixed_neigh_deg_seeds)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(q_array,avg_D, marker='o', label='Random')
plt.plot(q_array,avg_D_deg, marker='o', label='Degree')
plt.plot(q_array, avg_D_neigh_deg, marker='o', label='Neighbour degree')
plt.plot(q_array,avg_D_fixed_deg, marker='o', label='Fixed degree')
plt.plot(q_array,avg_D_fixed_neigh_deg, marker='o', label='Fixed neighbour degree')
plt.legend()
plt.xlabel('q')
plt.ylabel('Total Influenced Nodes')
plt.title('Average Influence Spread vs. q for different seeding protocol')
plt.show()
```



8: Computing time taken to form seed sets for each seeding protocol

Now for the same p_{seed}, network and q_array. We use python's **timeit** to determine the time taken to find the seeds in using each protocol.

In [ ]:
```
SETUP_CODE="""
from __main__ import degree_seeding
from __main__ import neighbour_degree_seeding
from __main__ import fixed_degree_seeding
from __main__ import fixed_neighbour_degree_seeding
import networkx as nx
import numpy as np
n=10000
m=10
p_seed=0.05
q_array = np.linspace(0.2,1.0,30)
G = nx.barabasi_albert_graph(n,m)"""

time_rand = timeit(stmt='np.random.choice(list({n for n in G}), size=int(p_seed*G.n
time_deg = timeit(stmt='degree_seeding(G, int(n*p_seed))',setup=SETUP_CODE, number=
time_neigh_deg = timeit(stmt='neighbour_degree_seeding(G, int(n*p_seed))',setup=SET
time_fixed_deg = timeit(stmt='fixed_degree_seeding(G, int(n*p_seed))',setup=SETUP_C
time_fixed_neigh_deg = timeit(stmt='fixed_neighbour_degree_seeding(G, int(n*p_seed)
```

In [ ]:
```
print('Time taken to form seeds for random seeding protocol is {}'.format(time_rand
print('Time taken to form seeds for degree seeding protocol is {}'.format(time_deg)
print('Time taken to form seeds for fixed degree seeding protocol is {}'.format(tim
print('Time taken to form seeds for neighbour degree seeding protocol is {}'.format
print('Time taken to form seeds for fixed neighbour degree seeding protocol is {}'.
```

```
Time taken to form seeds for random seeding protocol is 0.0269001999986358
Time taken to form seeds for degree seeding protocol is 0.12406659999396652
Time taken to form seeds for fixed degree seeding protocol is 23.103910400008317
Time taken to form seeds for neighbour degree seeding protocol is 3.326855199993588
Time taken to form seeds for fixed neighbour degree seeding protocol is 37.453471799
99226
```

Bonus I : Influence as a function of mean degree in graph

This is to show that it is imporant to keep m constant when generating a scale-free network, if there were correlation.

Generate graphs of various mean degrees and investigate influence/effective as a function of q.

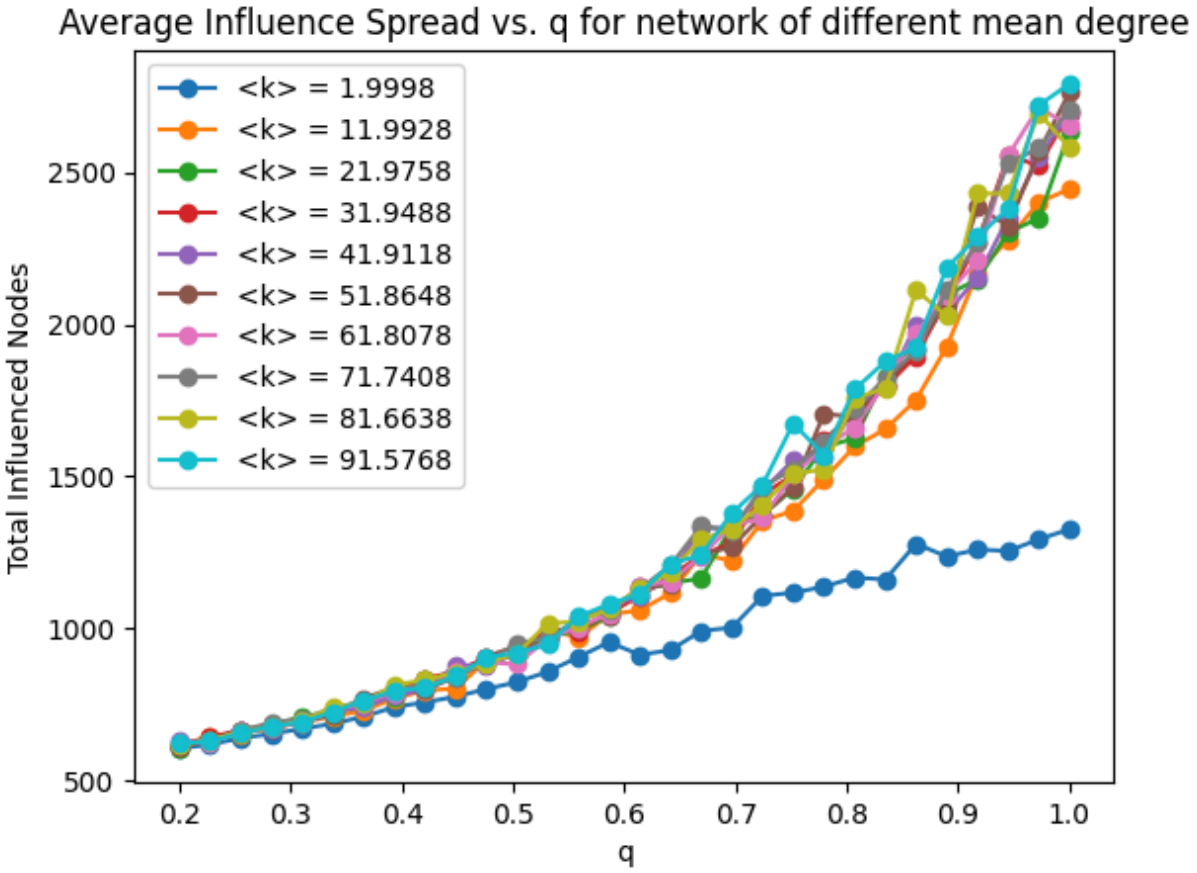
For a range of q (fix), generate graphs of various mean degree and investigate influence.

Is there a correlation? Does it plateau? Does the effective influence become 1 at different q values?

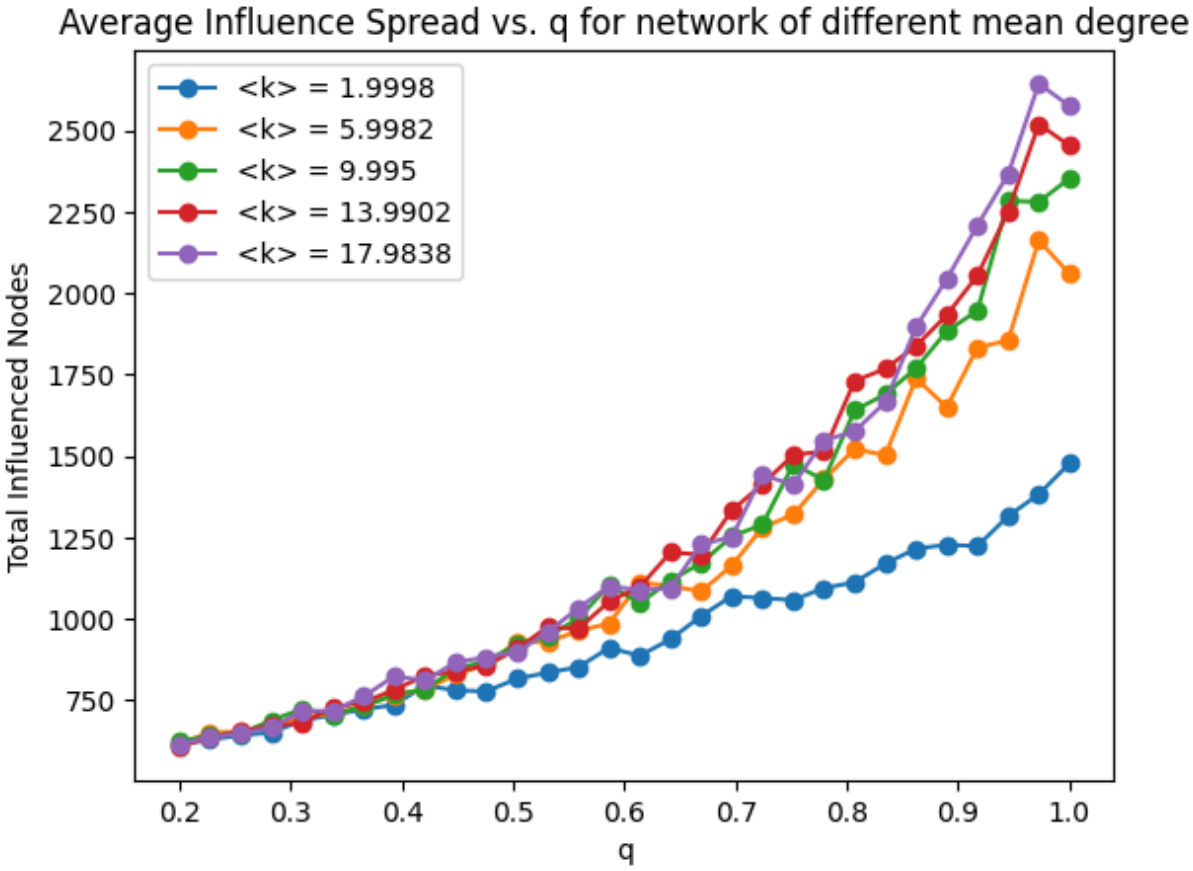
We can think of this as different marketing platforms. For example, trying to sell a product on Facebook could be different on TikTok because the m values on different platforms are

different.

In [ ]:
```
n = 10000
for m in range(1,50,5):

    G = nx.barabasi_albert_graph(n,m)
    mean_degree = np.mean([d for n,d in G.degree()])
    plt.plot(q_array, average_D_vs_q(G,q_array,None,p_seed=0.05,runs=5), marker='o'

plt.xlabel('q')
plt.ylabel('Total Influenced Nodes')
plt.title('Average Influence Spread vs. q for network of different mean degree')
plt.legend()
plt.show()
```



In [ ]:
```
n = 10000
for m in range(1,10,2):

    G = nx.barabasi_albert_graph(n,m)
    mean_degree = np.mean([d for n,d in G.degree()])
    plt.plot(q_array, average_D_vs_q(G,q_array,None,p_seed=0.05,runs=5), marker='o'

plt.xlabel('q')
plt.ylabel('Total Influenced Nodes')
plt.title('Average Influence Spread vs. q for network of different mean degree')
plt.legend()
plt.show()
```



Seems meaninful in picking m value ^

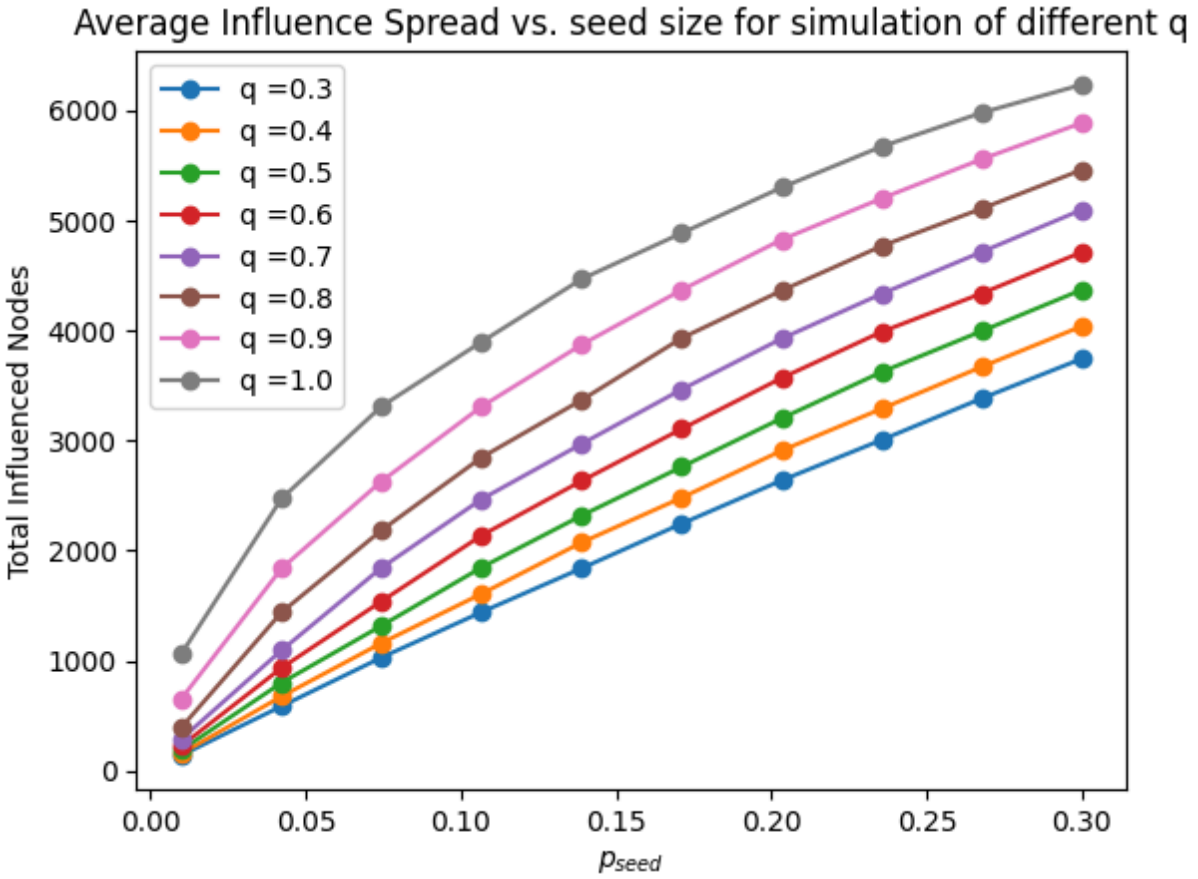
Bonus II : Influence as a function of initial seed size

For fixed q, fixed m=10, generate simulations of different initial seed size and final influence.

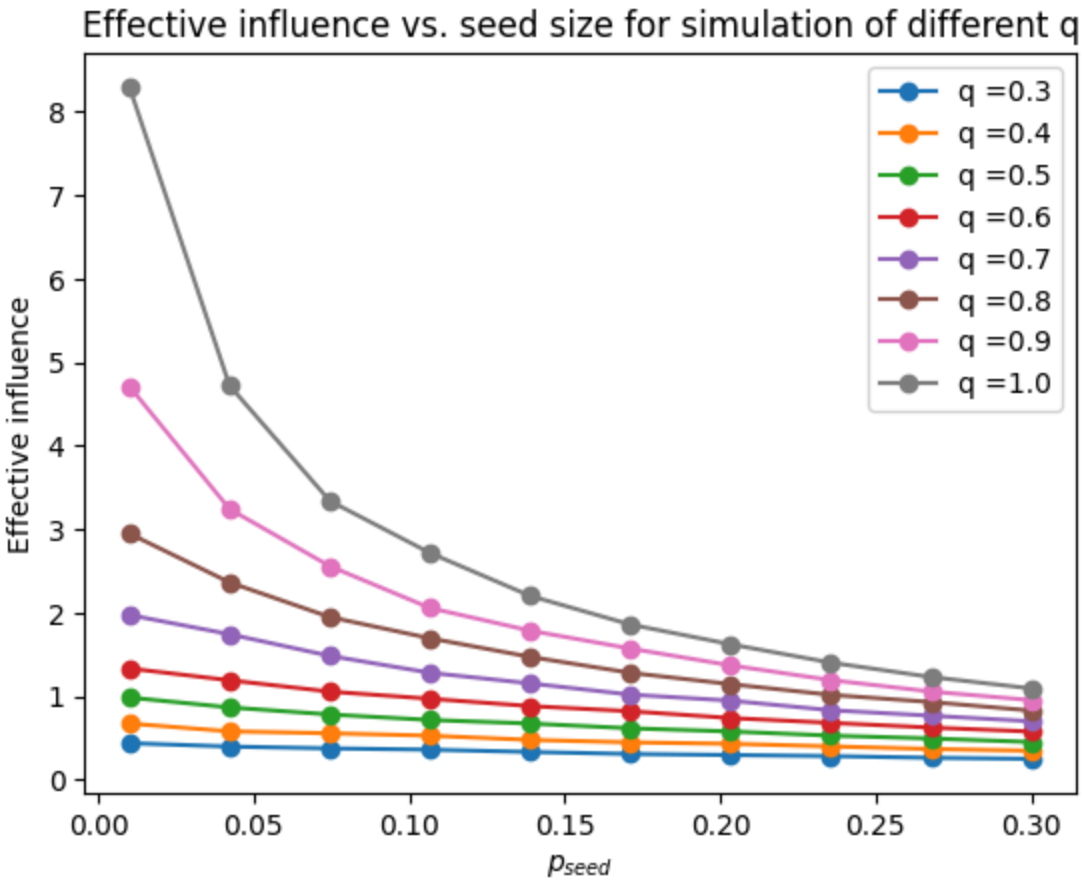
In [ ]:
```
def average_D_vs_p_seed(G, p_seed_array,q,initial_seeds=None,runs=20):
    avg_D = []

    for p_seed in p_seed_array:
        D_values = []
        for _ in range(runs):
            simulation = ICM_Model(G,q,initial_seeds,p_seed)
            I,A,D = simulation.run_to_extinction()
            D_values.append(D[-1])
        avg_D.append(np.mean(D_values))

    return avg_D
```

In [ ]:
```
q_array = [0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
p_seed_array = np.linspace(0.01,0.30,10)
n = 10000
m = 10
G = nx.barabasi_albert_graph(n,m)
```

```
for q in q_array:
        plt.plot(p_seed_array, average_D_vs_p_seed(G, p_seed_array, q), marker='o',

plt.xlabel(r'$p_{seed}$')
plt.ylabel('Total Influenced Nodes')
plt.title('Average Influence Spread vs. seed size for simulation of different q')
plt.legend()
plt.show()
```



In [ ]:
```
for q in q_array:
        plt.plot(p_seed_array, np.array(average_D_vs_p_seed(G, p_seed_array, q))/(n

plt.xlabel(r'$p_{seed}$')
plt.ylabel('Effective influence')
plt.title('Effective influence vs. seed size for simulation of different q')
plt.legend()
plt.show()
```



This seems meaningful ^

Bonus III : q vs. initial seed size

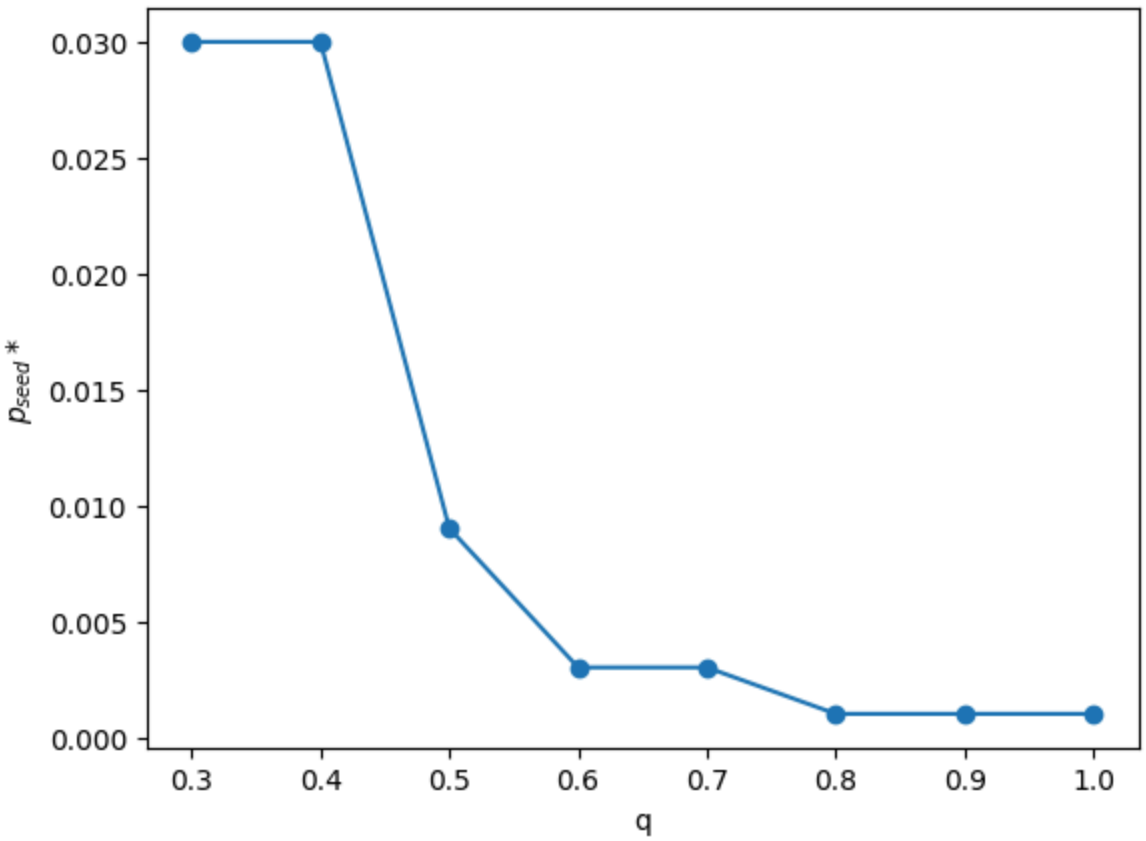
Different platforms have different q values, let's consider p_seed* to be the p_seed value for a network where effective influence = 1. What is the p_seed needed for effective influence to be 1 for a particular network and q ? The relationships shows eg. how many influencers does a company need to pay to reach a meaningful audience ? have a sizeable sale?

Estimate by numerical means and analytical means.

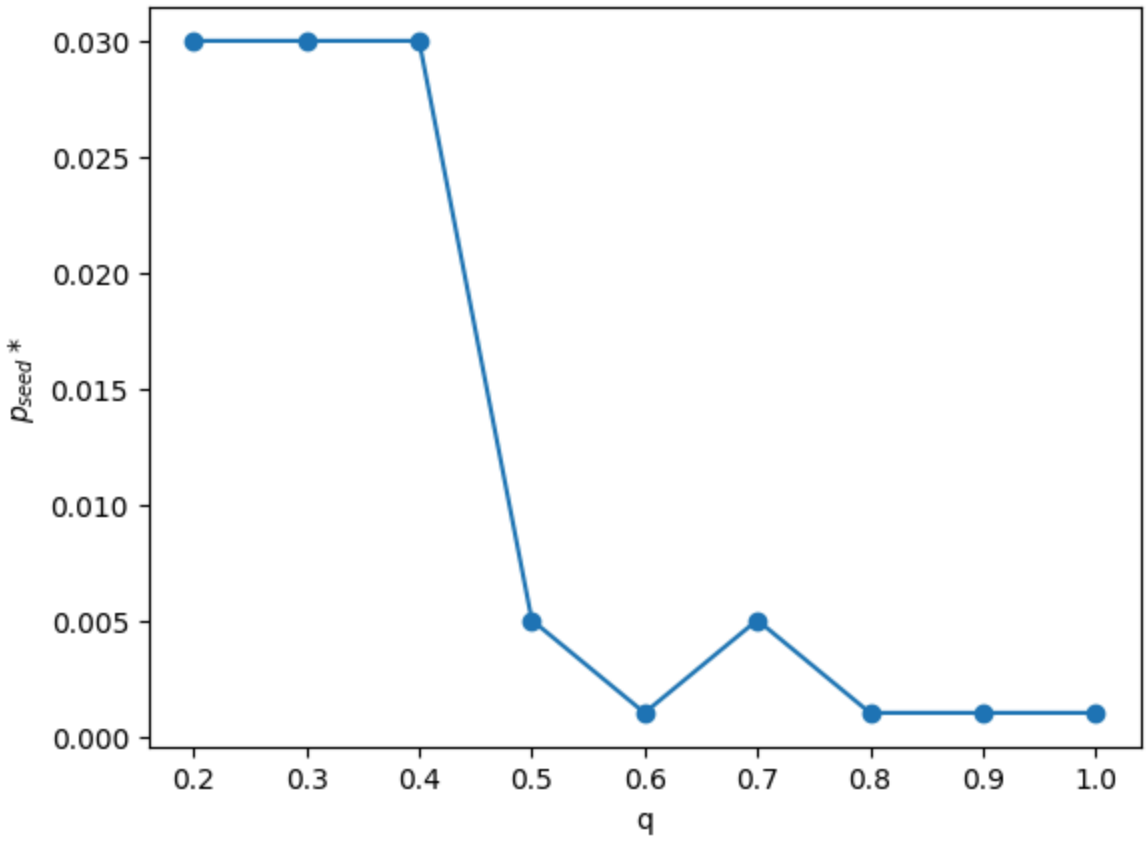
In [ ]:
```
def p_seed_thresh_for_q_array(G,q_array,runs=100):
    p_seed_array = np.linspace(0.001,1.0,500)
    ans = []
    for q in q_array:
        tmp = []
        for p_seed in p_seed_array:
            simulation = ICM_Model(G,q,None,p_seed)
            I,A,D = simulation.run_to_extinction()
            if (D[-1]/(p_seed*G.number_of_nodes()))>2:
                tmp.append(p_seed)
                break
        if tmp:
            ans.append(np.mean(tmp))
```

```
        else:
            ans.append(0.03)

    return ans
```

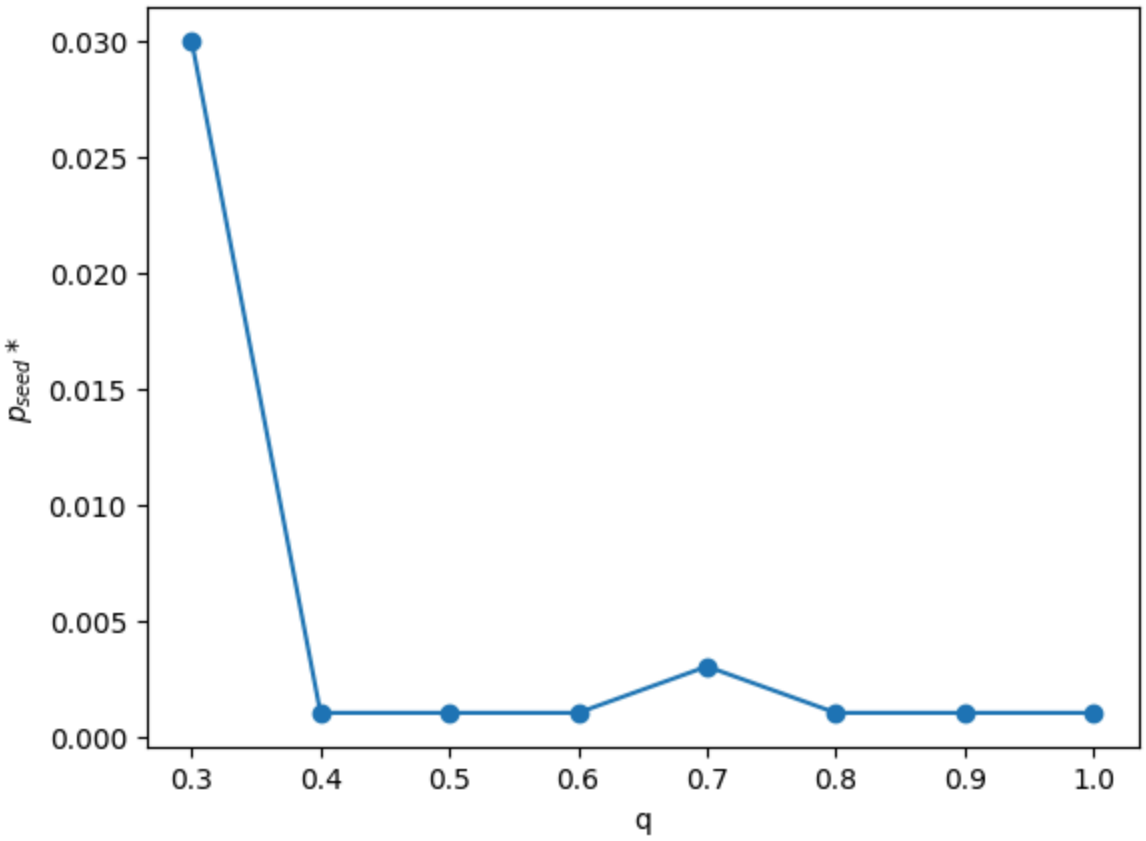
In [ ]:
```
n = 10000
m = 3

q_array = [0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]

plt.xlabel('q')
plt.ylabel(r'$p_{seed}*$')
plt.plot(q_array,p_seed_thresh_for_q_array(G,q_array),marker='o')
plt.show()
```



In [ ]:
```
n = 10000
m = 3

q_array = [0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]

plt.xlabel('q')
plt.ylabel(r'$p_{seed}*$')
plt.plot(q_array,p_seed_thresh_for_q_array(G,q_array),marker='o')
plt.show()
```



In [ ]:
```
n = 10000
m = 3

q_array = [0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]


plt.xlabel('q')
plt.ylabel(r'$p_{seed}*$')
plt.plot(q_array,p_seed_thresh_for_q_array(G,q_array),marker='o')
plt.show()
```



Just realised this is a pretty meaningless investigation... effective ratio>0 is like... a 100% bound to happen thing...

Analytical expression to determine influenced fraction

Let θ_i denote the probability that node i remains inactive after termination. The probability of node i never getting activated is none of its neighbours ever influencing it.

The probability of node j (neighbour of i) become activated and influencing node i is

$$(1-\theta_j)\frac{q}{k_i}$$

Therefore the probability of j never influencing i is 1 minus the above quantity

$$\theta_i = \prod_j (1-\frac{q(1-\theta_j)}{k_i})^{A_{ij}}$$

We can solve the equation above iteratively. Initialise θ_i to random values and iterate to convergence.

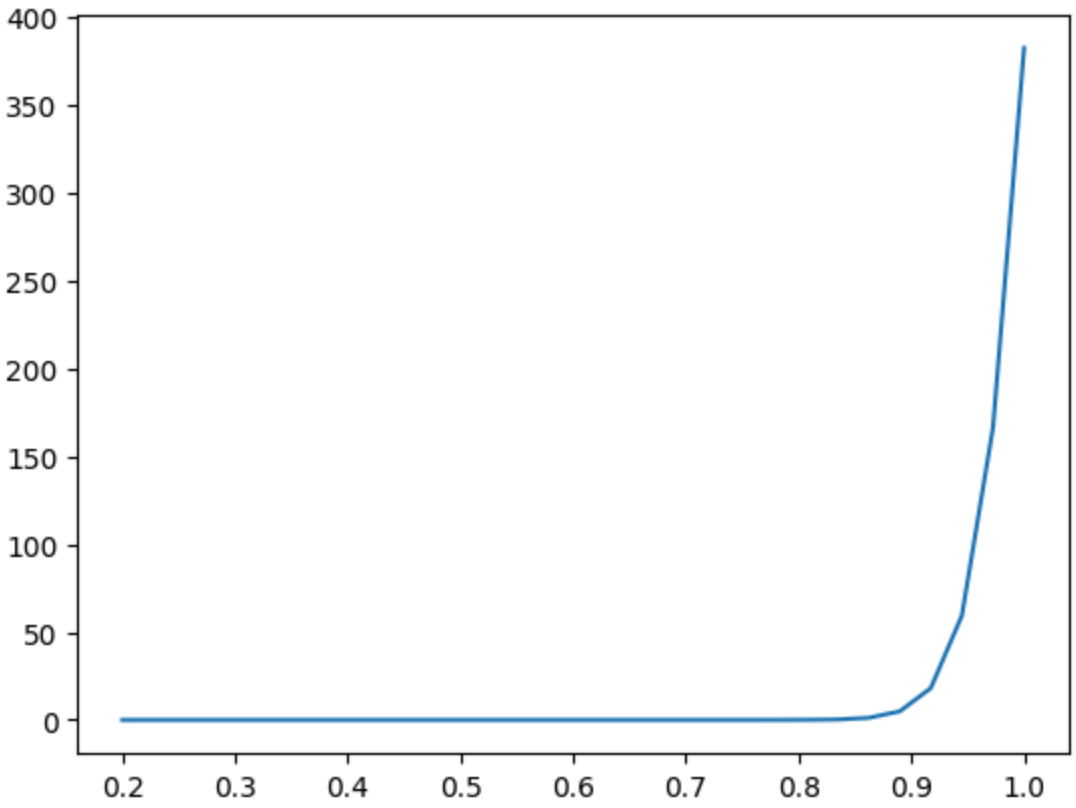
In [ ]:
```
def approx_D_prob_vs_q(G, q_array, tol=1e-6, max_iter=50):
    """Returns approximate average probability of final recovered size by iterative
```

```
    approx_D = []

    for q in q_array:
        t = np.random.uniform(size=G.number_of_nodes())

        for _ in range(50):
            t_new = np.ones(G.number_of_nodes())
            for i in range(G.number_of_nodes()):
                t_new[i] = np.prod([1-q*(1-t[j])/(G.degree[i]) for j in G.neighbors
            if np.linalg.norm(t_new-t)<1e-6:
                break
            t = t_new

        approx_D.append(1-np.mean(t))

    return approx_D
```

In [ ]:
```
n = 10000
m = 10
q_array = np.linspace(0.2,1.0,30)
G = nx.barabasi_albert_graph(n,m)
approx_D_prob = approx_D_prob_vs_q(G,q_array)
```

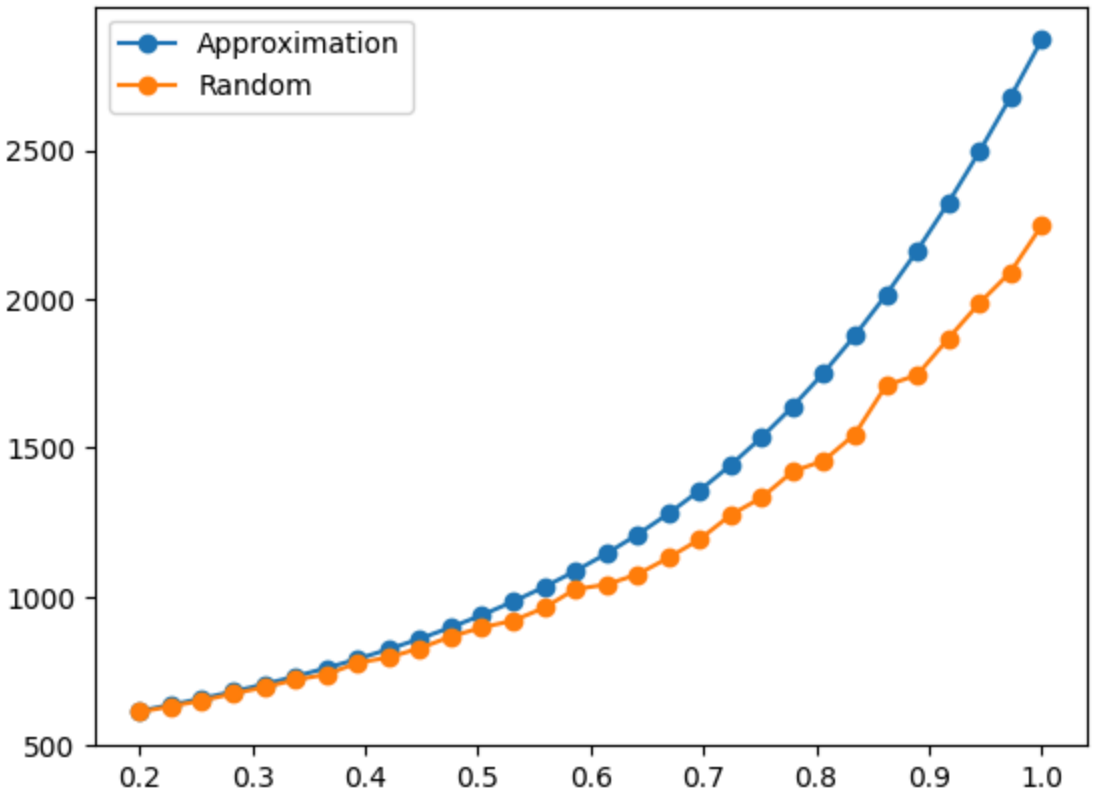
```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[296], line 5
      3 q_array = np.linspace(0.2,1.0,30)
      4 G = nx.barabasi_albert_graph(n,m)
----> 5 approx_D_prob = approx_D_prob_vs_q(G,q_array)

Cell In[295], line 11, in approx_D_prob_vs_q(G, q_array, tol, max_iter)
      9 t_new = np.ones(G.number_of_nodes())
     10 for i in range(G.number_of_nodes()):
---> 11     t_new[i] = np.prod([1-q*(1-t[j])/(G.degree[i]) for j in G.neighbors(i)])
     12 if np.linalg.norm(t_new-t)<1e-6:
     13     break

Cell In[295], line 11, in <listcomp>(.0)
      9 t_new = np.ones(G.number_of_nodes())
     10 for i in range(G.number_of_nodes()):
---> 11     t_new[i] = np.prod([1-q*(1-t[j])/(G.degree[i]) for j in G.neighbors(i)])
     12 if np.linalg.norm(t_new-t)<1e-6:
     13     break

KeyboardInterrupt:
```

In [ ]:
```
plt.plot(q_array, np.array(approx_D_prob)*n)
plt.show()
```



In [ ]:
```
def gpt_sol(p_seed,q_array):
    a = np.random.uniform()
    arr=[]

    for q in q_array:
        for _ in range(50):
            a_new = p_seed + (1-p_seed)*(1-np.exp(-q*a))
            if np.linalg.norm(a_new-a)<1e-6:
                break
            a = a_new
        arr.append(a)

    return arr
```

In [ ]:
```
plt.plot(q_array,np.array(gpt_sol(0.05,q_array))*10000,marker='o', label='Approxima
plt.plot(q_array,avg_D, marker='o', label='Random')
plt.legend()
plt.show()
```



Given that a node of degree k, is activated, the expected number of node it will influence is

$$E(Y|k) = k\sum_{k'} P(k'|k)\frac{q}{k'}$$

Denote $p(k)$ as the probability of a node with degree k occuring, then the expected no. of influence for a activated node is simply :

$$E(Y) = \sum_{k} E(Y|k)p(k)$$
$$= \sum_{k} p(k)k\sum_{k'} P(k'|k)\frac{q}{k'}$$

Denote α as mean probability of a node being active.

The number of active nodes we expect in a network can therefore be

$$\alpha * n = n * p_{seed} + p_{seed} * n * E(Y) + p_{seed} * n * E(Y)^2 + \ldots p_{seed} * n * E(Y)^{longestpath}$$

Simplifying the expression gives

$$\alpha = p_{seed}\sum_{i=0}^{D} E(Y)^i$$

Where D is the diameter of the network.

For Barabasi-Albert network, the degree distribution of a neighbour of a node of degree k is

$$p(l|k) = \frac{m(k+2)}{kl(l+1)}(1 - \frac{math.comb(2m+2, m+1) * math.comb(k+l-2m, l-m)}{math.comb(k+l+2, l+1)})$$