In [1]:
```
#Importing all the required libraries
import random
import pickle
```

In [2]:
```
#Generating weights for all items
#weights is a dictionary, keys->items, values-> weights
weights=dict()
for i in range(500):
    weights[i]=random.uniform(0,2) #weight is a random value betwee
n 0 and 2
```

In [3]:
```
#Generating prices for all items
#prices is a dictionary, keys-> items, values->prices
prices=dict()
for i in range(500):
    prices[i]=random.randint(100,1000)#Since we are generating cart
s ourselves we have generated prices randomly
```

In [4]:
```
#Storing the data in pickle files to maintain uniformity across dif
ferent implementations
file_weights = 'weights'
outfile1 = open(file_weights,'wb')
pickle.dump(weights,outfile1)
outfile1.close()
```

In [5]:
```
#Storing the data in pickle files to maintain uniformity across dif
ferent implementations
file_prices = 'prices'
outfile2 = open(file_prices,'wb')
pickle.dump(prices,outfile2)
outfile2.close()
```

In [6]:
```
#Generating 1000 carts for 10 experiments with 500 products
carts=dict()
for experiment in range(10):
    current_carts=[]
    for cart in range(1000):
        new_cart=[]
        for product in range(500):
            prob=random.uniform(0,1)
            if(prob<=0.02):
                new_cart.append(product)
        current_carts.append(new_cart)
    carts[experiment]=current_carts
```

In [7]:
```
#Storing the data in pickle files to maintain uniformity across dif
ferent implementations
file_carts = 'carts'
outfile3 = open(file_carts,'wb')
pickle.dump(carts,outfile3)
outfile3.close()
```

In [8]:
```
#Reading the amazon co-purchasing data. Since the amazon data has v
ery large no. of nodes we have filtered it
#to consider only 500 nodes
#The data is in the form of edge lists
amazon_500=[]
file = open("amazon.txt")
for line in file.readlines():
    if(line[0]=='#'):
        continue
    else:
        line=line.rstrip("\n")
        line=line.split("\t")
        from_node=int(line[0])
        to_node=int(line[1])
        if(from_node>=500):
            break
        if(to_node<500):
            amazon_500.append((from_node,to_node))
```

In [9]:
```
#BFS Traversal of a network
#Inputs-> source node, total no. of nodes, adjacency list dictionar
y of a network
#BFS follow a FIFO traversal.
#This method returns a list. index-> Node no. Value-> Shortest dist
ance between node and source node
def bfs(s,n,adj_list):

    visited = [False] * (n) #Visited stores the status of each node
. Initially all nodes are unvisited
    temp_list=dict()#Stores the distance of all the nodes from the
source node
    queue = [] #BFS uses a queue data structure. Queue follows the
FIFO property
    queue.append(s)
    visited[s] = True
    temp_list[s]=0
    while queue:
            # Dequeue a vertex from the queue.This node becomes the
current node
        node = queue.pop(0)
            #Now we need to visit all the unvisited neighbouring no
des of the current node
            #distance of these unvisited neighbouring nodes would b
e distance of current_node+1
        for i in adj_list[node]:

            if visited[i] == False:
                queue.append(i)
                visited[i] = True
                temp_list[i]=temp_list[node]+1

    return temp_list
```

In [10]:
```
#Average Path Length/Characteristic Path Length
#Path Length= Length of shortest path between 2 nodes.
#BFS returns the shortest path between 2 nodes
#Avg Path Length= sum(shortest path length between all unique pairs
)/no. of unique pairs
#No. of unique pairs=(no. of nodes * no. of nodes-1)/2
#This method takes the adj list dictionary of the network as input
#This method returns the characteristic path length of the network
def avg_path_length(n,adj_list):
    distance_list=[]
    for s in range(n):  #Calling bfs by considering each node as a
source node
        temp_list=bfs(s,n,adj_list)
    distance_list.append(temp_list)
    avg_path_length=0
    #Summing up the distances between all the node pairs
    for curr_dict in distance_list:
        for distances in curr_dict.values():
            avg_path_length+=distances
    avg_path_length=avg_path_length/2 #Need to count distance betwe
en any 2 nodes only once
    avg_path_length=2*avg_path_length/(n*(n-1))
    return avg_path_length
```

In [11]:
```
#Converting the amazon edge list to adjacency list
adj_list_amazon=dict()
for i in range(500):
    adj_list_amazon[i]=[]
for edge in amazon_500:
    node1=edge[0]
    node2=edge[1]
    if(node2 not in adj_list_amazon[node1]):
        adj_list_amazon[node1].append(node2)
    if(node1 not in adj_list_amazon[node2]):
        adj_list_amazon[node2].append(node1)
```

In [12]:
```
#Calculating average path length for amazon co-purchasing data
avg_path_length_amazon=avg_path_length(500,adj_list_amazon)
print(avg_path_length_amazon)
```

```
0.009322645290581162
```

In [13]:
```
#Creating an empty list of dicts
#Index of the list represents experiment no.
#Value is an empty dict representing the adj list
#key->product no
#value->empty list to hold the neighbours
list_of_adj_lists=[]
for experiment in range(10):
    print(experiment)
    adj_list=dict()
    for product in range(500):
        adj_list[product]=[]
    list_of_adj_lists.append(adj_list)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [14]:
```
#Calculating the avg path lengths for all the 10 experiments
#Updating the list_of_adj_lists to hold actual adj list which are g
enerated from the carts
avg_path_length_list=[]
for experiment in range(10):
    adj_list=list_of_adj_lists[experiment]
    cart_lists=carts[experiment]
    for cart in cart_lists:
        for i in range(len(cart)-1):
            for j in range(i+1,len(cart),1):
                if(cart[j] not in adj_list[cart[i]]):
                    adj_list[cart[i]].append(cart[j])
                if(cart[i] not in adj_list[cart[j]]):
                    adj_list[cart[j]].append(cart[i])
    avg_path_length_list.append((avg_path_length(500,adj_list)))
```

In [15]:
```
#Average path length across all the 10 experiments
print(sum(avg_path_length_list)/10)
```

```
0.0033414829659318634
```

In [16]:
```
#Sorting the neighbours list in all adj lists
for experiment in range(10):
    adj_list=list_of_adj_lists[experiment]
    for product in range(500):
        adj_list[product].sort()
```

In [17]:
```
#Storing the data in pickle files to maintain uniformity across dif
ferent implementations
file_adj_list = 'adj_list'
outfile6 = open(file_adj_list,'wb')
pickle.dump(list_of_adj_lists,outfile6)
outfile6.close()
```

In [18]:
```
#Calculating f and g
# f represents the frequency score of an item
# g represents the frequecy score of 2 items purchased together
#initializing f and g's to 0
f=[]
g=[]
for experiment in range(10):
    temp_dict=dict()
    for product in range(500):
        temp_dict[product]=0
    f.append(temp_dict)
for experiment in range(10):
    temp_dict_g=dict()
    for product1 in range(499):
        for product2 in range(product1,500,1):
            temp_dict_g[(product1,product2)]=0
    g.append(temp_dict_g)
```

In [19]:
```
#Calculating the values of f and g
for experiment in range(10):
    carts_list=carts[experiment]
    for cart in carts_list:
        if(len(cart)==1):
            f[experiment][cart[0]]+=1
        else:
            k=len(cart)
            for i in range(len(cart)-1):
                current_product=cart[i]
                f[experiment][current_product]+=(1/k)
                for j in range(i+1,len(cart),1):
                    copurchase_product=cart[j]
                    g[experiment][(current_product,copurchase_produ
ct)]+=(1/k)
```

In [20]:
```
#Storing the data in pickle files to maintain uniformity across dif
ferent implementations
file_f = 'f'
outfile4 = open(file_f,'wb')
pickle.dump(f,outfile4)
outfile4.close()
```

In [21]:
```
#Storing the data in pickle files to maintain uniformity across dif
ferent implementations
file_g = 'g'
outfile5 = open(file_g,'wb')
pickle.dump(g,outfile5)
outfile5.close()
```