Justin Estaris - 100528069

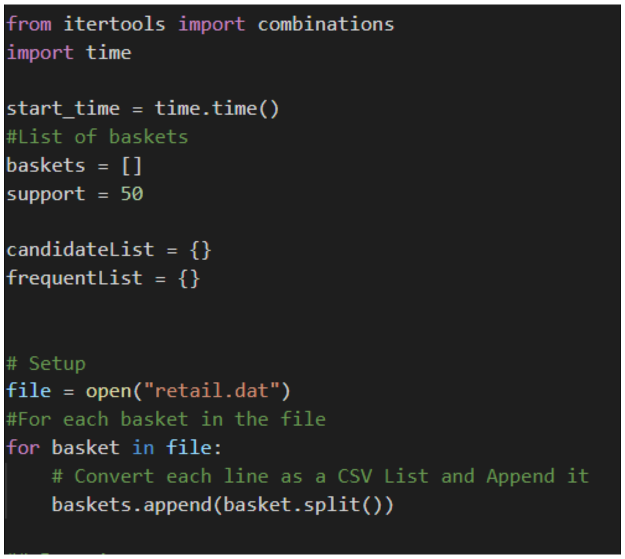
Apriori Algorithm

Code breakdown

At the start of code I imported the combinations function from itertools and time and started the time function at the right after. I then initialized a couple of variables for storage which do the following:

- Baskets – A list that will contain each row as a basket from the dataset
- Support – A variable used to filter out non-frequent items
- candidateList – A dictionary used to contain all candidate items
- frequentList – A dictionary used to contain all frequent items

After initialization, the program then opens the file called retail.dat. The program will then convert each line into a comma separated value list and add each line to the baskets list. The result will give me a list of lists.



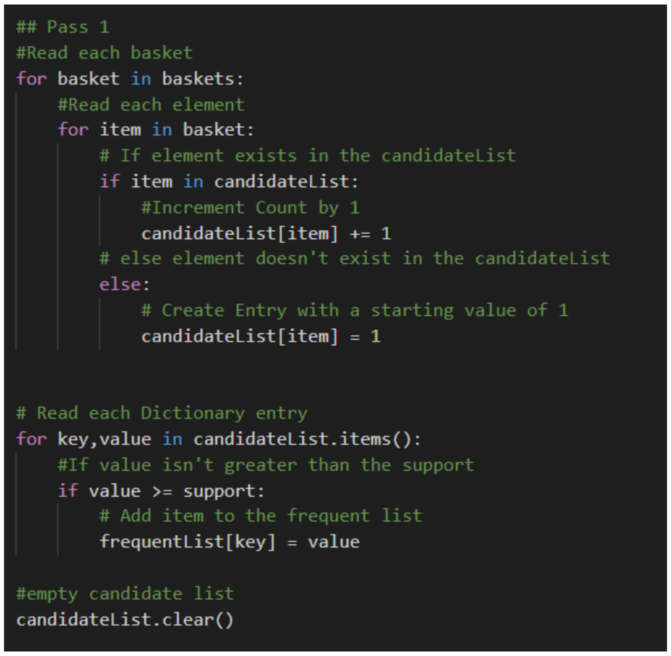
```
from itertools import combinations
import time

start_time = time.time()
#List of baskets
baskets = []
support = 50

candidateList = {}
frequentList = {}


# Setup
file = open("retail.dat")
#For each basket in the file
for basket in file:
    # Convert each line as a CSV List and Append it
    baskets.append(basket.split())
```

Figure 1. Apriori Start

Pass 1

In the first pass of Apriori the program will loop over every basket that's in the basket list and for every basket that's looped over the program will also loop over every item that each basket has. The program will then check if the item is in the candidate list (which is represented by a dictionary). If the item is already in the candidate list increase the count by one. If it doesn't exist however, create a new entry with the item name as the key and give it a starting value of one.

After looping over each item in every basket, the program will then determine if the item is considered a frequent item. To do this, the program will use the candidate list it generated earlier and compare its count to the support value that was initialized at the beginning. If the count is greater than or equal to the support, it will create a new entry with the item name as its key and count as its value. Afterwards, the program will then clear the candidate list in preparation for the second pass.



```
## Pass 1
#Read each basket
for basket in baskets:
    #Read each element
    for item in basket:
        # If element exists in the candidateList
        if item in candidateList:
            #Increment Count by 1
            candidateList[item] += 1
        # else element doesn't exist in the candidateList
        else:
            # Create Entry with a starting value of 1
            candidateList[item] = 1


# Read each Dictionary entry
for key,value in candidateList.items():
    #If value isn't greater than the support
    if value >= support:
        # Add item to the frequent list
        frequentList[key] = value

#empty candidate list
candidateList.clear()
```

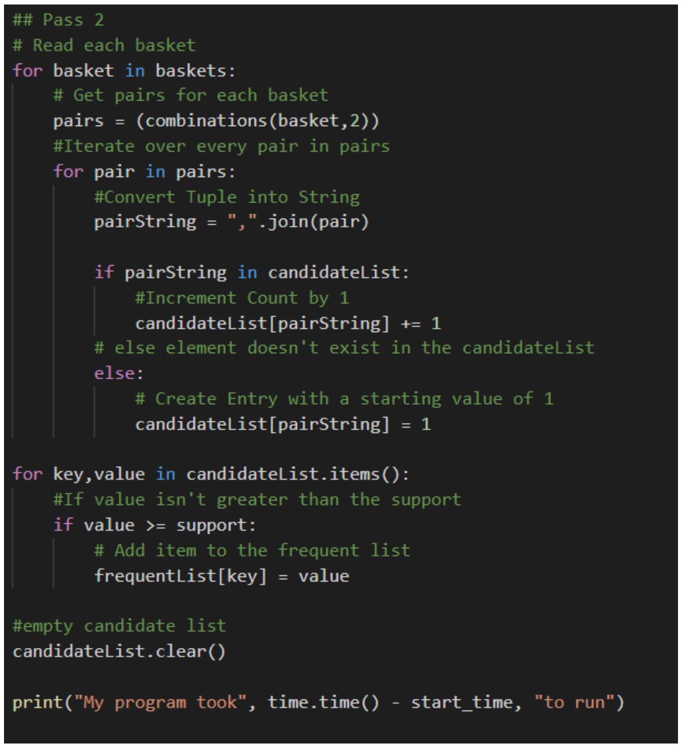
Figure 2. Apriori Pass 1

Pass 2

Similar to pass one, the program will iterate over each basket in the baskets list. However, instead of iterating over each element in the list the program will generate all the possible pairs from each basket. This is done by using the combinations function that is provided from itertools. The program then iterates over each pair, converts it into a string, and then determines if the pair is in the candidate list. If it is it will increment its count by one. If not, it will create a new entry with its name as the key and with count value starting at one.

After looping over each basket and creating each pair possible from each of them, the program then determines if the pair is a frequent item. It does this by using the candidate list if created earlier and compares each count the by its support value. If the count is greater than or equal to the support, it will create a new entry with the item name as its key and count as its value.

In the end, the program will then clear the candidate list and will print out the duration of the program.

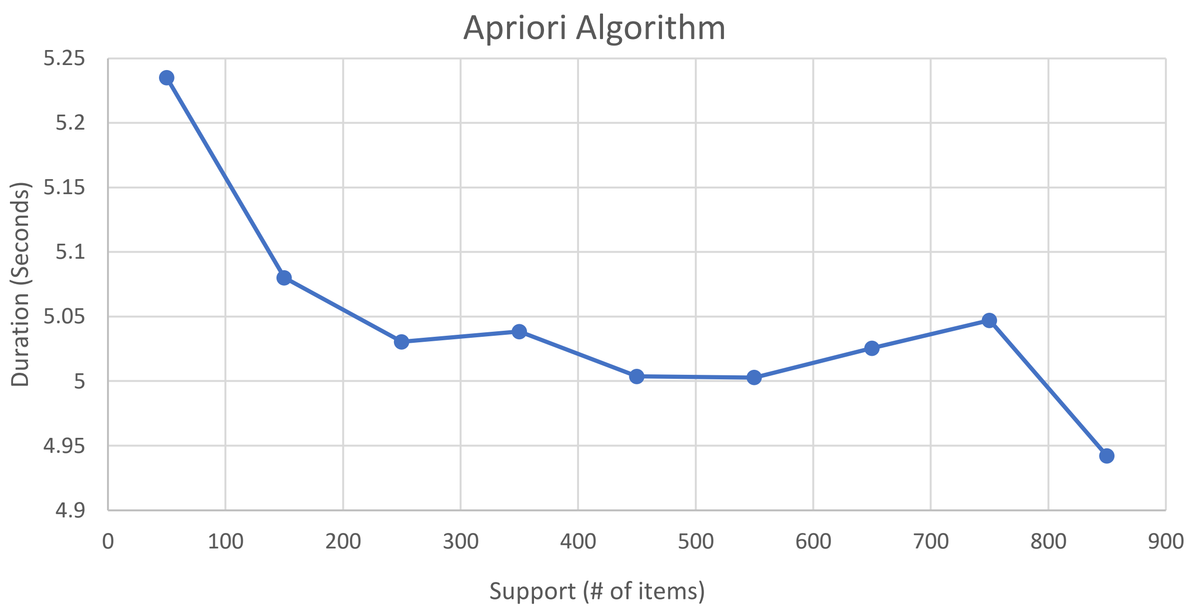


```python
## Pass 2
# Read each basket
for basket in baskets:
    # Get pairs for each basket
    pairs = (combinations(basket,2))
    #Iterate over every pair in pairs
    for pair in pairs:
        #Convert Tuple into String
        pairString = ",".join(pair)

        if pairString in candidateList:
            #Increment Count by 1
            candidateList[pairString] += 1
        # else element doesn't exist in the candidateList
        else:
            # Create Entry with a starting value of 1
            candidateList[pairString] = 1

for key,value in candidateList.items():
    #If value isn't greater than the support
    if value >= support:
        # Add item to the frequent list
        frequentList[key] = value

#empty candidate list
candidateList.clear()

print("My program took", time.time() - start_time, "to run")
```

Figure 3. Apriori Pass 2

Analysis

Graph: Support % vs. Time + Discussion



In the graph above as we increase the support for the dataset, the duration of the program decreases. This is because at the end of pass one the algorithm will have reduced the amount of singletons due to failing the support test. Since the amount of singletons has been reduced the amount of pairs generated by program is also reduced which in turn reduces the amount of checks needed to determine if the candidate item is a frequent item. Thus resulting in a shorter duration.

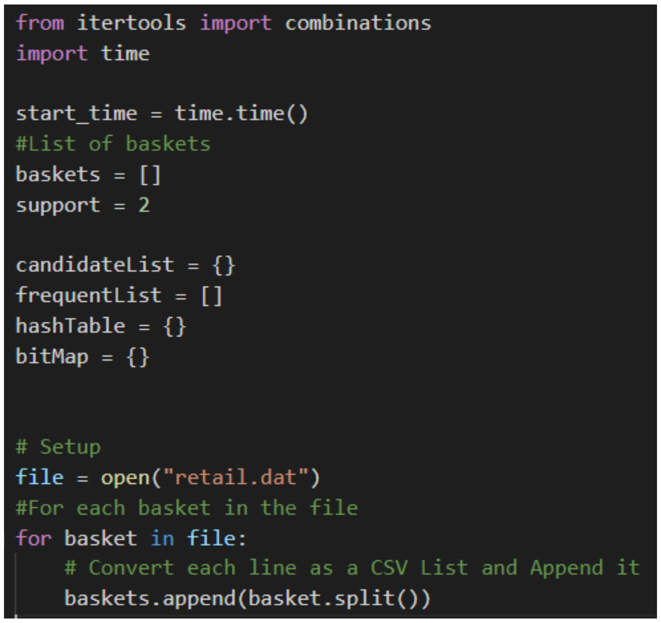
PCY Algorithm

Code overview

what is different from previous implementation?

Similar to the Apriori algorithm the purpose of the PCY Algorithm is to determine frequent item sets from a dataset. The main difference however is that we use the idle memory on the first pass to create a hash table by hashing each possible pair from each basket we iterate over. After the first pass, the program will then convert the hash table into a binary map where if the hash bucket is frequent it will receive a value of one, if not a zero. The second pass is also similar to Apriori, the main difference is that checks the pair's singleton values, to see if they're frequent, and if it hashes to a frequent bucket. If it the pair meets both of those conditions the program will add it to the frequent item list.

The start of the program is very similar to Apriori in terms of code, the main difference is that is has hash table (to store all the possible hashes and its count) and a bit map.



```
from itertools import combinations
import time

start_time = time.time()
#List of baskets
baskets = []
support = 2

candidateList = {}
frequentList = []
hashTable = {}
bitMap = {}


# Setup
file = open("retail.dat")
#For each basket in the file
for basket in file:
    # Convert each line as a CSV List and Append it
    baskets.append(basket.split())
```

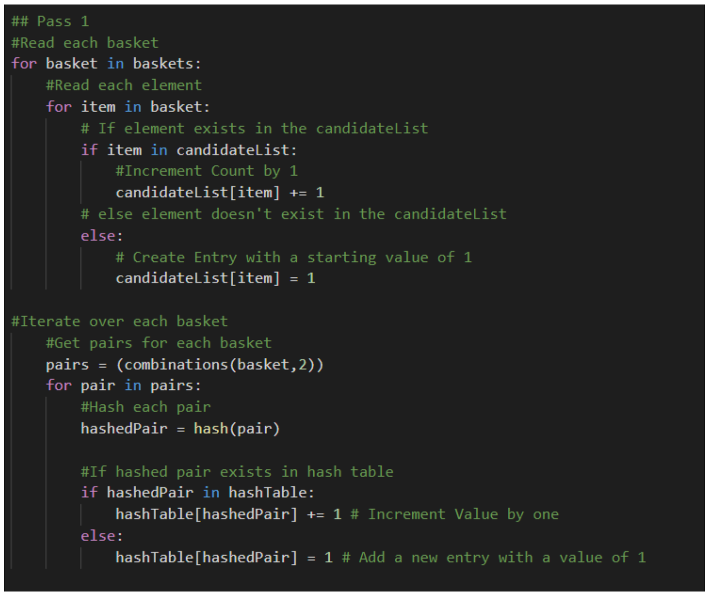
Figure 4. PCY Start

Discussion on Hash Function

The hash function used in the program is the built-in function for Python 3.x. The program uses this to decrease the amount of main memory it will use for storage. The more main memory the program can use the shorter the duration.

Pass 1

Similar to Apriori in terms of code, but the main difference is the additional loop after iterating over each item in the basket. This for-loop will create all possible pairs from the basket, hash the pair and will add it to the hash table with a starting count-value of one. If the pair exists on the hash table, it will increment the count by 1

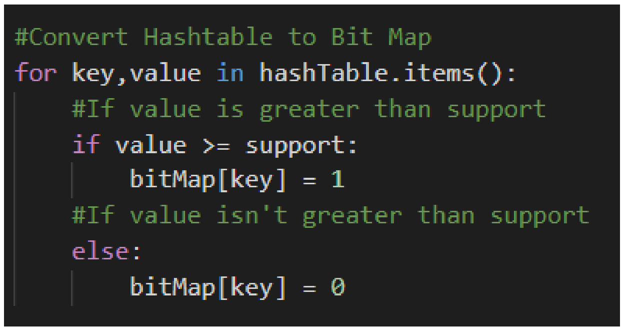


```
## Pass 1
#Read each basket
for basket in baskets:
    #Read each element
    for item in basket:
        # If element exists in the candidateList
        if item in candidateList:
            #Increment Count by 1
            candidateList[item] += 1
        # else element doesn't exist in the candidateList
        else:
            # Create Entry with a starting value of 1
            candidateList[item] = 1

#Iterate over each basket
    #Get pairs for each basket
    pairs = (combinations(basket,2))
    for pair in pairs:
        #Hash each pair
        hashedPair = hash(pair)

        #If hashed pair exists in hash table
        if hashedPair in hashTable:
            hashTable[hashedPair] += 1 # Increment Value by one
        else:
            hashTable[hashedPair] = 1 # Add a new entry with a value of 1
```

Figure 5. PCY Pass 1

In-between Step

After creating the hash table the program is only needs to know if a hash is frequent and not its count. So it will iterate over each hash table item and compare its value to the support. If the value is greater than or equal to the support, we add the key to the bitmap with a value of 1. If not, we add the key to the bitmap with a value of 0.

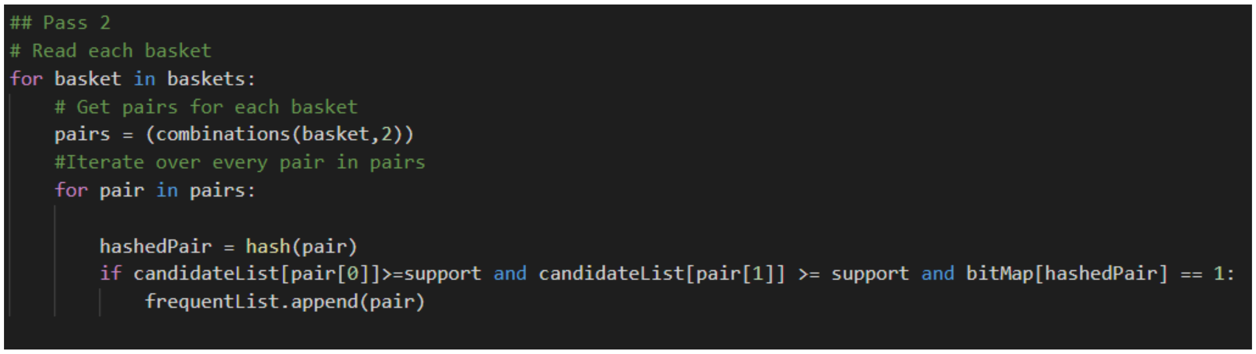


```
#Convert Hashtable to Bit Map
for key,value in hashTable.items():
    #If value is greater than support
    if value >= support:
        bitMap[key] = 1
    #If value isn't greater than support
    else:
        bitMap[key] = 0
```

Figure 6. PCU In-between Step

Pass 2

In Pass two, the program will iterate each basket in the basket list and create pairs from each basket. Afterwards, the program will then hash the pair and store its hash for the check. If the first element, second element and hashed value are considered frequent then the program will add the pair to the frequent list.

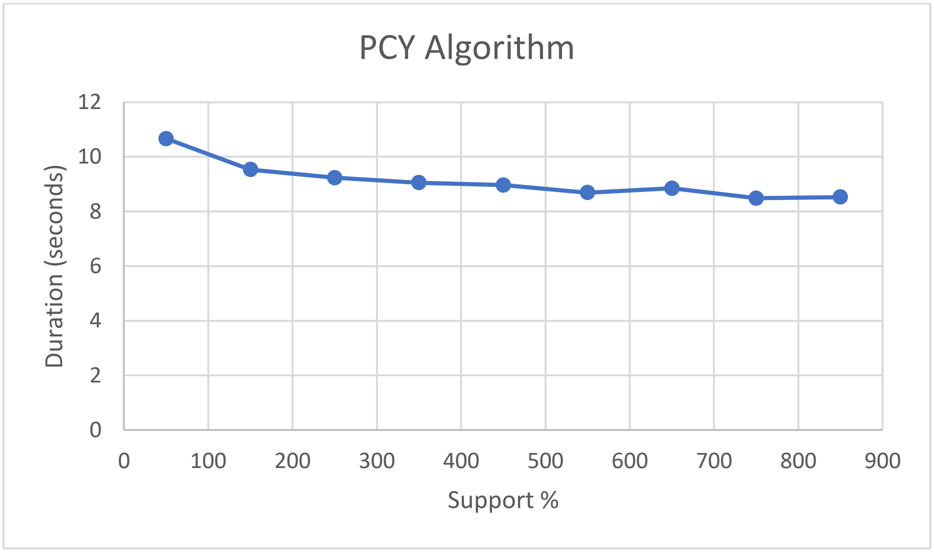


```
## Pass 2
# Read each basket
for basket in baskets:
    # Get pairs for each basket
    pairs = (combinations(basket,2))
    #Iterate over every pair in pairs
    for pair in pairs:

        hashedPair = hash(pair)
        if candidateList[pair[0]]>=support and candidateList[pair[1]] >= support and bitMap[hashedPair] == 1:
            frequentList.append(pair)
```

Figure 7. Pass 2

Analysis

Graph: Support % vs. Time + Discussion



In the graph above you can see that as we increase the support the duration of the PCY Algorithm is also shortened. The difference is slight because the amount of candidate pairs that passes the conditions for the

Buckets Number vs Time + Discussion

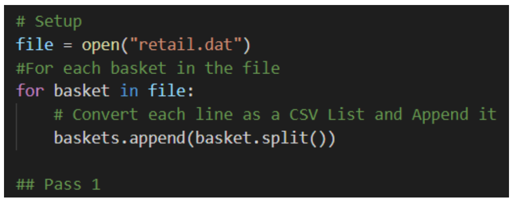
Optimal buckets number

# of Pairs = # of buckets for Optimal Time

Random Sampling & SON

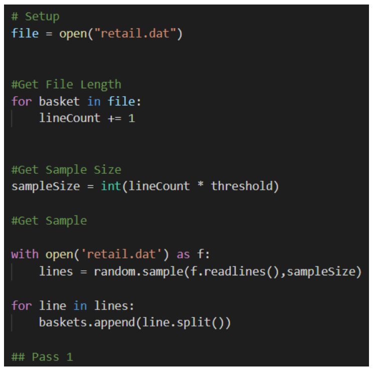
Explain the difference between your Random Sampling Implementation and Apriori

The main difference between my implantation between Random Sampling and Apriori is the amount of rows/baskets I'll be using for my analysis. In the Apriori Algorithm, the program will use the whole data-set which will increase the duration, but have an accurate result. Whereas, Random sampling will only take a portion of the dataset (at random) which will result in a shorter duration, but less than accurate result. In my case my code will only take in the amount of rows multiplied by the threshold as my sample size. The analysis process remains the same as the Apriori algorithm.



```
# Setup
file = open("retail.dat")
#For each basket in the file
for basket in file:
    # Convert each line as a CSV List and Append it
    baskets.append(basket.split())

## Pass 1
```

Apriori Set up



```
# Setup
file = open("retail.dat")


#Get File Length
for basket in file:
    lineCount += 1


#Get Sample Size
sampleSize = int(lineCount * threshold)

#Get Sample

with open('retail.dat') as f:
    lines = random.sample(f.readlines(),sampleSize)

for line in lines:
    baskets.append(line.split())

## Pass 1
```

Random Sampling Set up

What is the ideal sampling % and strategy?

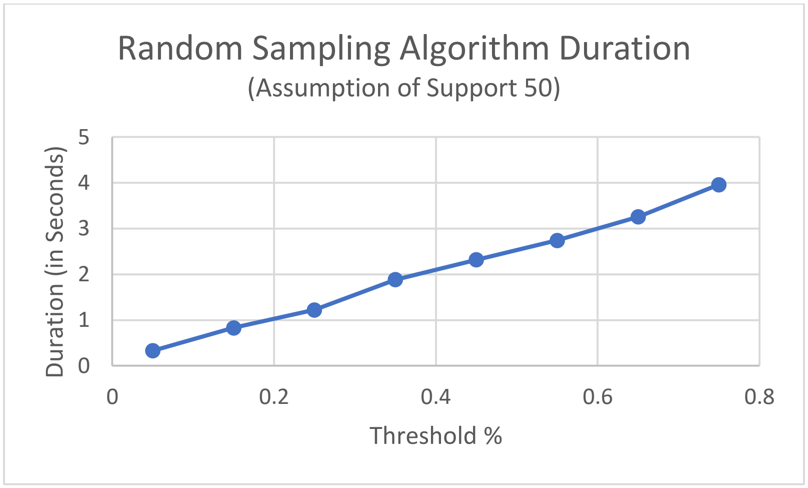


Figure 8. Random Sampling - Threshold Changes

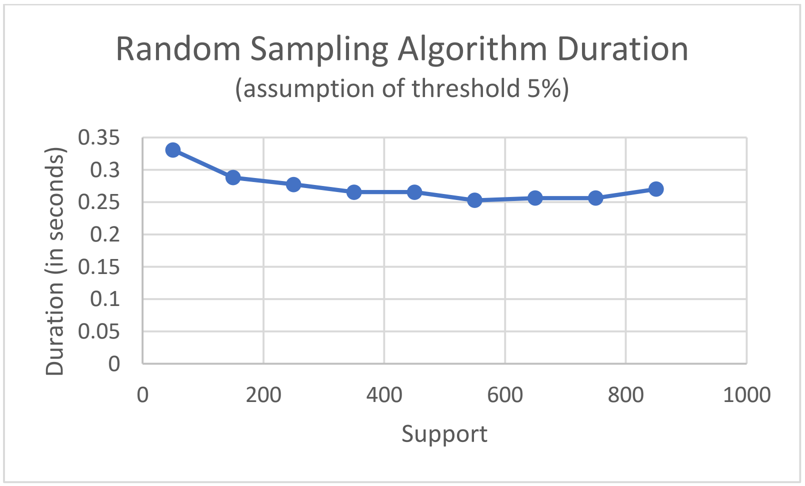
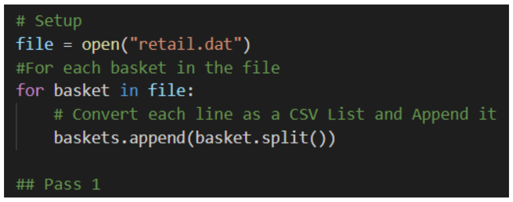


Figure 9.Random Sample - Support Changes

On Figure 8 above we can see that as we increase the sample size threshold, the longer the duration of the program. Additionally, in Figure 9. As the program increase the support count the duration decreases. Looking at both of these graphs, the ideal sample % and strategy

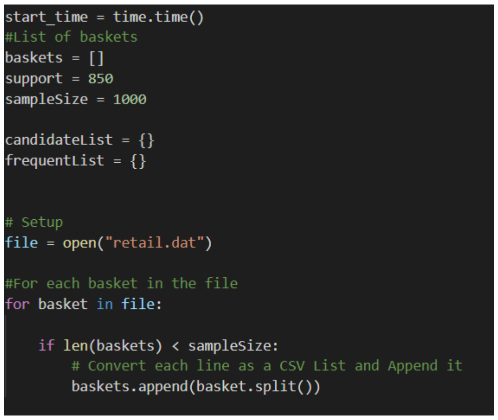
Explain the difference between your SON Implementation and Apriori

The difference between my SON and Apriori Implementation is that instead of taking in the whole file as one big set (which is what Apriori does), I split the document into smaller pieces (called chunks) to feed and analyze that dataset. My implementation takes splits the document every 1,000 rows. I take advantage of monotonicity (where if the item is frequent in the chunk, then it's frequent in the whole document) to speed up the analyzation process which results in the shorter duration times.



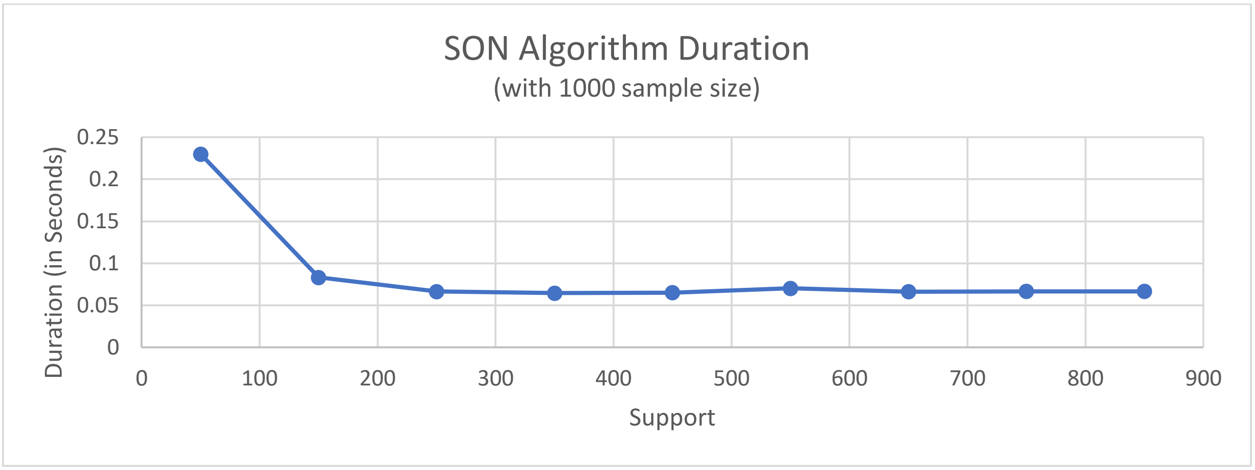
```
# Setup
file = open("retail.dat")
#For each basket in the file
for basket in file:
    # Convert each line as a CSV List and Append it
    baskets.append(basket.split())

## Pass 1
```

Apriori Set up



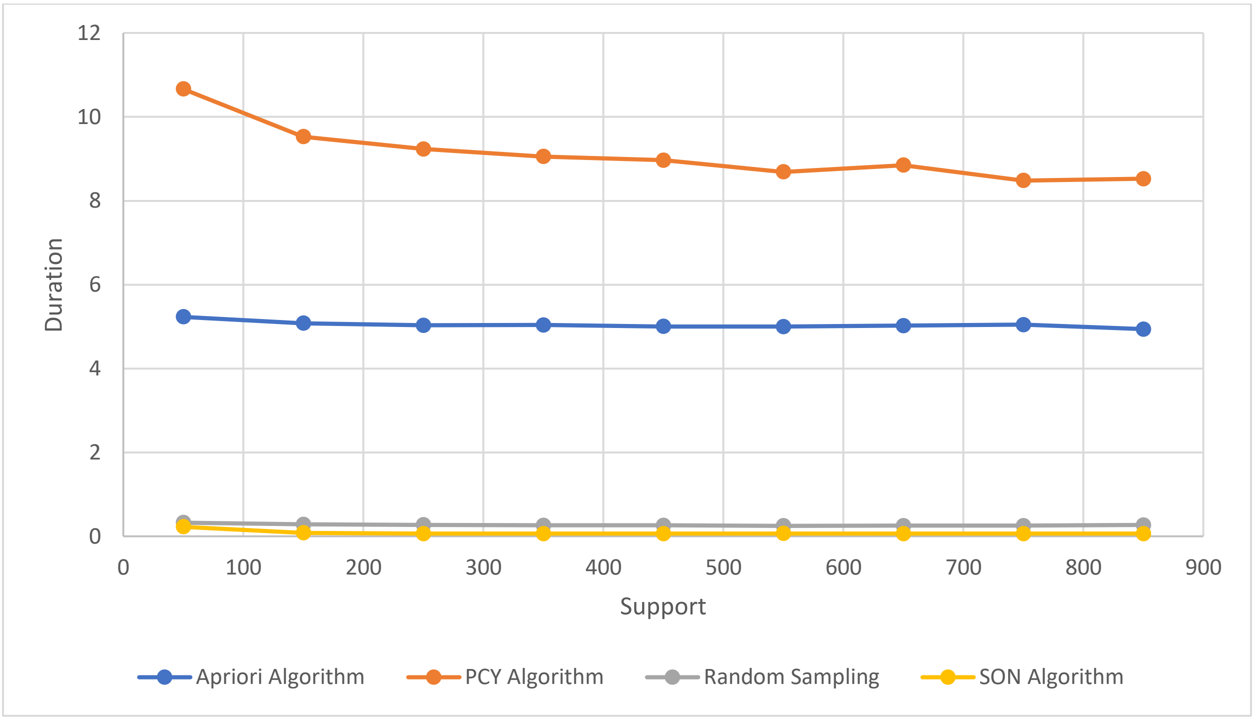
```
start_time = time.time()
#List of baskets
baskets = []
support = 850
sampleSize = 1000

candidateList = {}
frequentList = {}


# Setup
file = open("retail.dat")

#For each basket in the file
for basket in file:

    if len(baskets) < sampleSize:
        # Convert each line as a CSV List and Append it
        baskets.append(basket.split())
```

SON Set up



Justin Estaris - 100528069

Apriori vs. PCY vs. RS vs. SON



Discuss why one algorithm might be faster than the other, and in what cases

Based on the graph we can see that PCY is the slowest algorithm on duration compared to the other algorithms. This because the PCY algorithm did not eliminate 75% of the candidate items. Since it doesn't eliminate that 75% candidate items then it will do extra work compared to the Apriori algorithm.

Random Sampling and the SON Algorithm are the fastest algorithms with SON being the fastest, by only a slight margin. These are considered the fastest because they are only taking a small portion of the whole document. So it's trading off accuracy with speed.

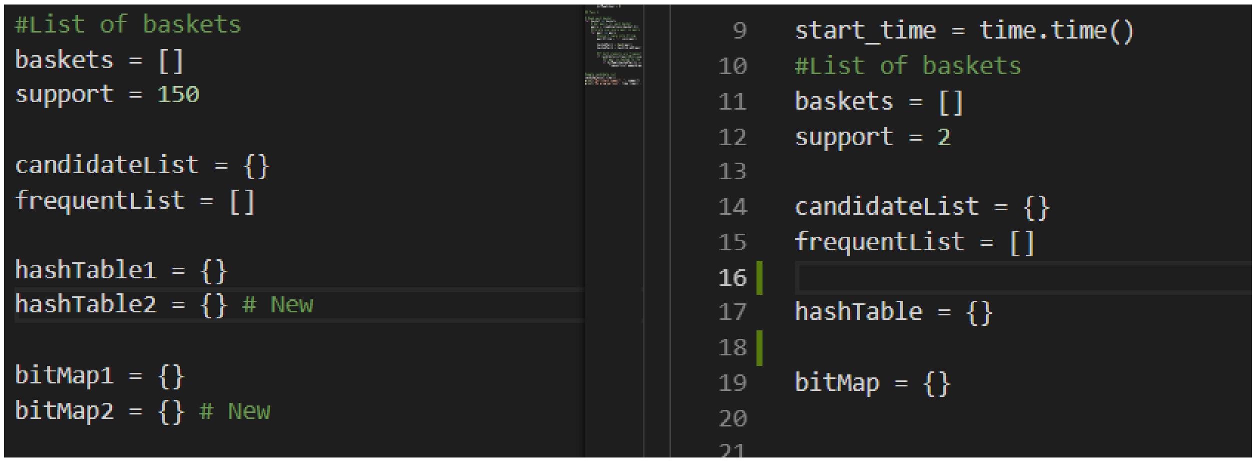
Optional

Multi-hash Algorithm

The optional implementation I chose was the Multi-Hash algorithm.

How does this algorithm differ from your original PCY implementation?

This algorithm differs from my original PCY implementation by adding an extra hashing, bitmap functionality and one additional step.

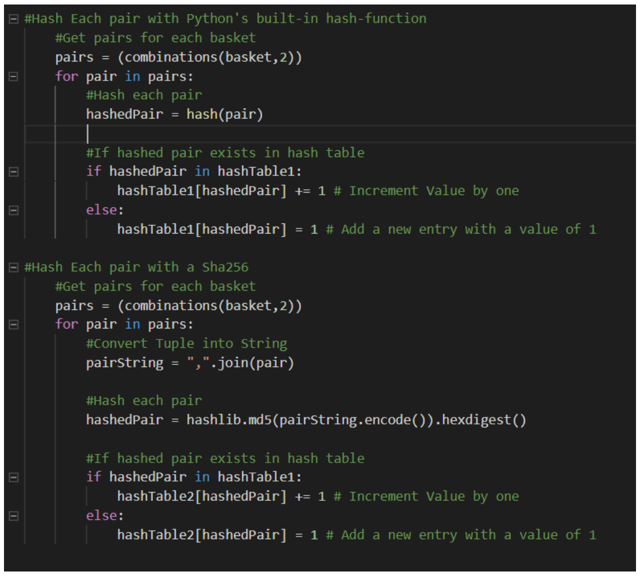


```
#List of baskets
baskets = []
support = 150

candidateList = {}
frequentList = []

hashTable1 = {}
hashTable2 = {} # New

bitMap1 = {}
bitMap2 = {} # New
```

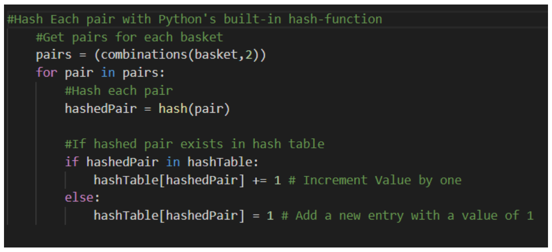
```
start_time = time.time()
#List of baskets
baskets = []
support = 2

candidateList = {}
frequentList = []

hashTable = {}

bitMap = {}
```

Figure 10. Mutlihash vs PCY Implementation

In the above figure, I've added an extra hash table and bitmap which will be used in both passes.



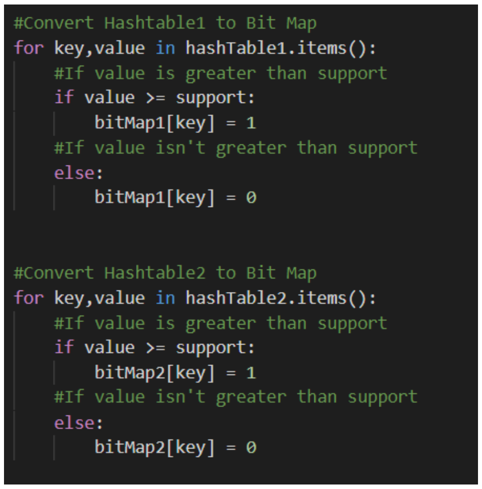
```
#Hash Each pair with Python's built-in hash-function
    #Get pairs for each basket
    pairs = (combinations(basket,2))
    for pair in pairs:
        #Hash each pair
        hashedPair = hash(pair)

        #If hashed pair exists in hash table
        if hashedPair in hashTable1:
            hashTable1[hashedPair] += 1 # Increment Value by one
        else:
            hashTable1[hashedPair] = 1 # Add a new entry with a value of 1

#Hash Each pair with a Sha256
    #Get pairs for each basket
    pairs = (combinations(basket,2))
    for pair in pairs:
        #Convert Tuple into String
        pairString = ",".join(pair)

        #Hash each pair
        hashedPair = hashlib.md5(pairString.encode()).hexdigest()

        #If hashed pair exists in hash table
        if hashedPair in hashTable1:
            hashTable2[hashedPair] += 1 # Increment Value by one
        else:
            hashTable2[hashedPair] = 1 # Add a new entry with a value of 1
```



```
#Hash Each pair with Python's built-in hash-function
    #Get pairs for each basket
    pairs = (combinations(basket,2))
    for pair in pairs:
        #Hash each pair
        hashedPair = hash(pair)

        #If hashed pair exists in hash table
        if hashedPair in hashTable:
            hashTable[hashedPair] += 1 # Increment Value by one
        else:
            hashTable[hashedPair] = 1 # Add a new entry with a value of 1
```

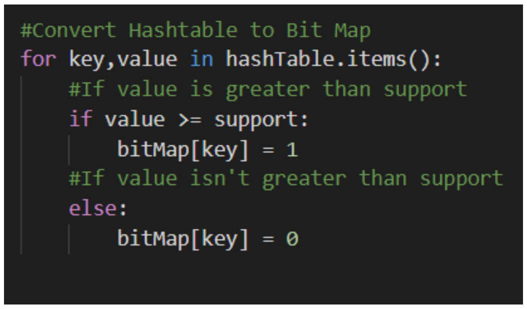
Multihash Implementation　　　　*PCY Implementation*

In the above pictures I've added an extra for loop that will not only hash the pair with Python's built-in hash function, but also hash the pair to an MD5 hash.



```
#Convert Hashtable1 to Bit Map
for key,value in hashTable1.items():
    #If value is greater than support
    if value >= support:
        bitMap1[key] = 1
    #If value isn't greater than support
    else:
        bitMap1[key] = 0


#Convert Hashtable2 to Bit Map
for key,value in hashTable2.items():
    #If value is greater than support
    if value >= support:
        bitMap2[key] = 1
    #If value isn't greater than support
    else:
        bitMap2[key] = 0
```

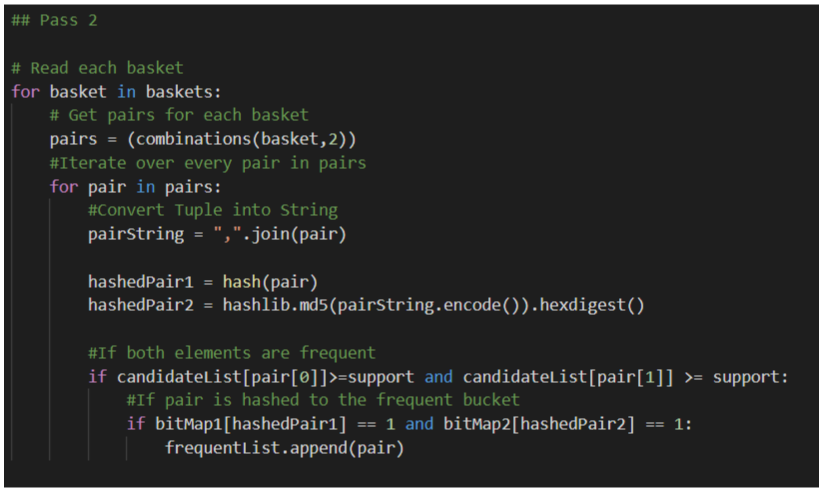
Multihash Implementation



```
#Convert Hashtable to Bit Map
for key,value in hashTable.items():
    #If value is greater than support
    if value >= support:
        bitMap[key] = 1
    #If value isn't greater than support
    else:
        bitMap[key] = 0
```

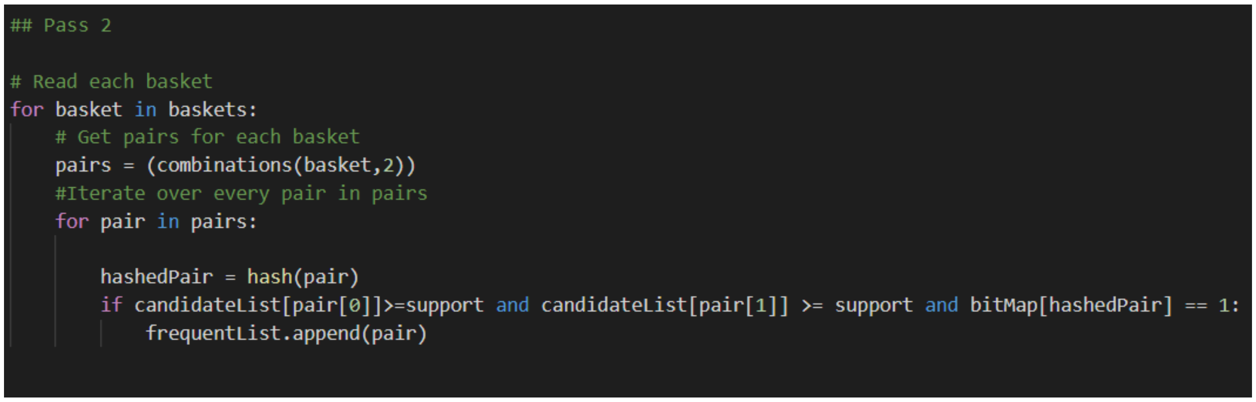
PCY Implementation

Additionally, the conversion from hash table to bitmap process remains the same, but done twice with the Multihash Algorithm.



```
## Pass 2

# Read each basket
for basket in baskets:
    # Get pairs for each basket
    pairs = (combinations(basket,2))
    #Iterate over every pair in pairs
    for pair in pairs:
        #Convert Tuple into String
        pairString = ",".join(pair)

        hashedPair1 = hash(pair)
        hashedPair2 = hashlib.md5(pairString.encode()).hexdigest()

        #If both elements are frequent
        if candidateList[pair[0]]>=support and candidateList[pair[1]] >= support:
            #If pair is hashed to the frequent bucket
            if bitMap1[hashedPair1] == 1 and bitMap2[hashedPair2] == 1:
                frequentList.append(pair)
```

Figure 11. Multihash Pass 2



```
## Pass 2

# Read each basket
for basket in baskets:
    # Get pairs for each basket
    pairs = (combinations(basket,2))
    #Iterate over every pair in pairs
    for pair in pairs:

        hashedPair = hash(pair)
        if candidateList[pair[0]]>=support and candidateList[pair[1]] >= support and bitMap[hashedPair] == 1:
            frequentList.append(pair)
```

Figure 12. PCY Pass 2

In pass two, similar to the PCY algorithm, it checks to see if the pair's singletons are considered frequent items and if the pair hashes to a frequent bucket for bitmap1. With Multi-hash it does an additional check to see if it also hashes to a frequent bucket for bitmap2. These two checks minimize the amount of false-positives in the set.

What are the different hash functions and how many functions/stages were used?

The different hash functions I've used for the multi-hash algorithm is Python's built-in hash function and the MD5 Hash. Similar to the PCY Algorithm there are two stages/passes and no additional functions have been added to the algorithm.