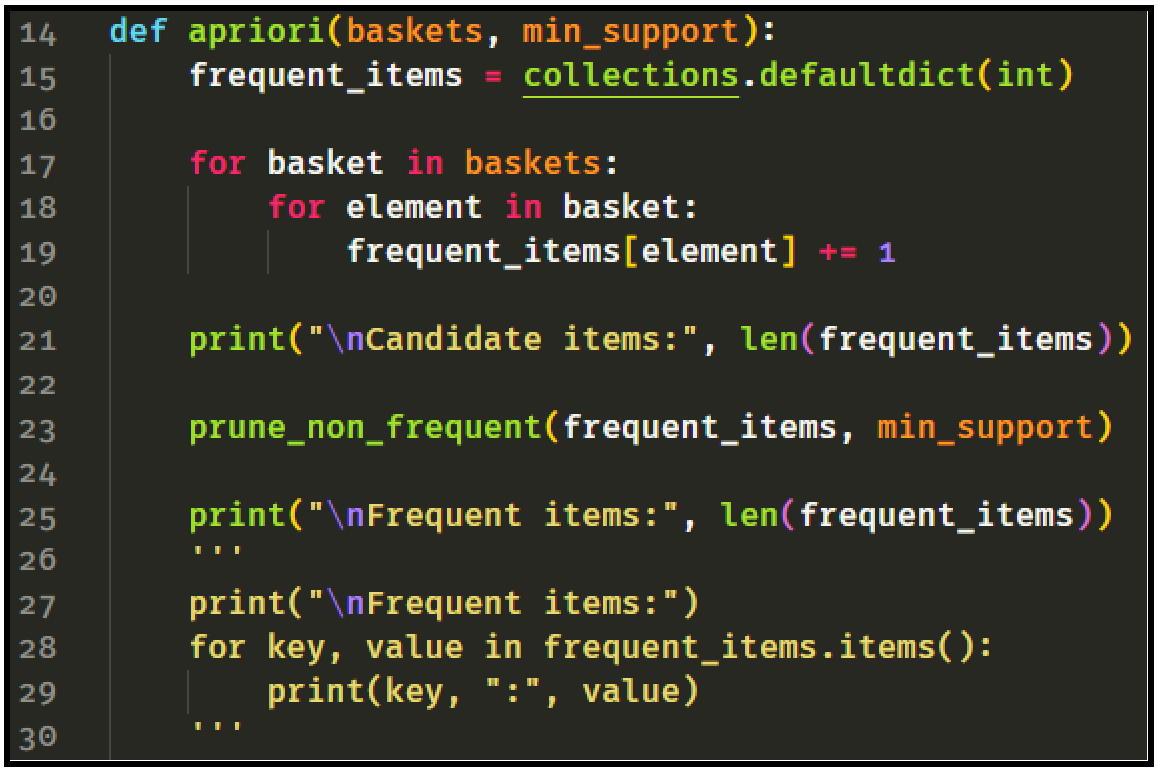
Movie Recommendation Algorithm

(Finding Frequent Item Sets)

Apriori:

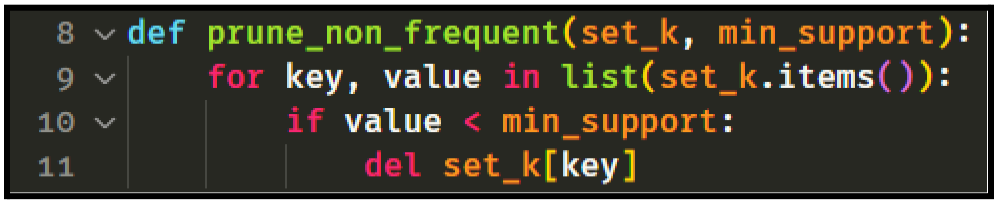
- **Code breakdown:**

  Pass 1: Read the baskets and count the occurrences of all items in the dataset**.** Prune all items that appear less times than the support number (s). The items passed the support threshold are the frequent items.



```
14  def apriori(baskets, min_support):
15      frequent_items = collections.defaultdict(int)
16
17      for basket in baskets:
18          for element in basket:
19              frequent_items[element] += 1
20
21      print("\nCandidate items:", len(frequent_items))
22
23      prune_non_frequent(frequent_items, min_support)
24
25      print("\nFrequent items:", len(frequent_items))
26      '''
27      print("\nFrequent items:")
28      for key, value in frequent_items.items():
29          print(key, ":", value)
30      '''
```

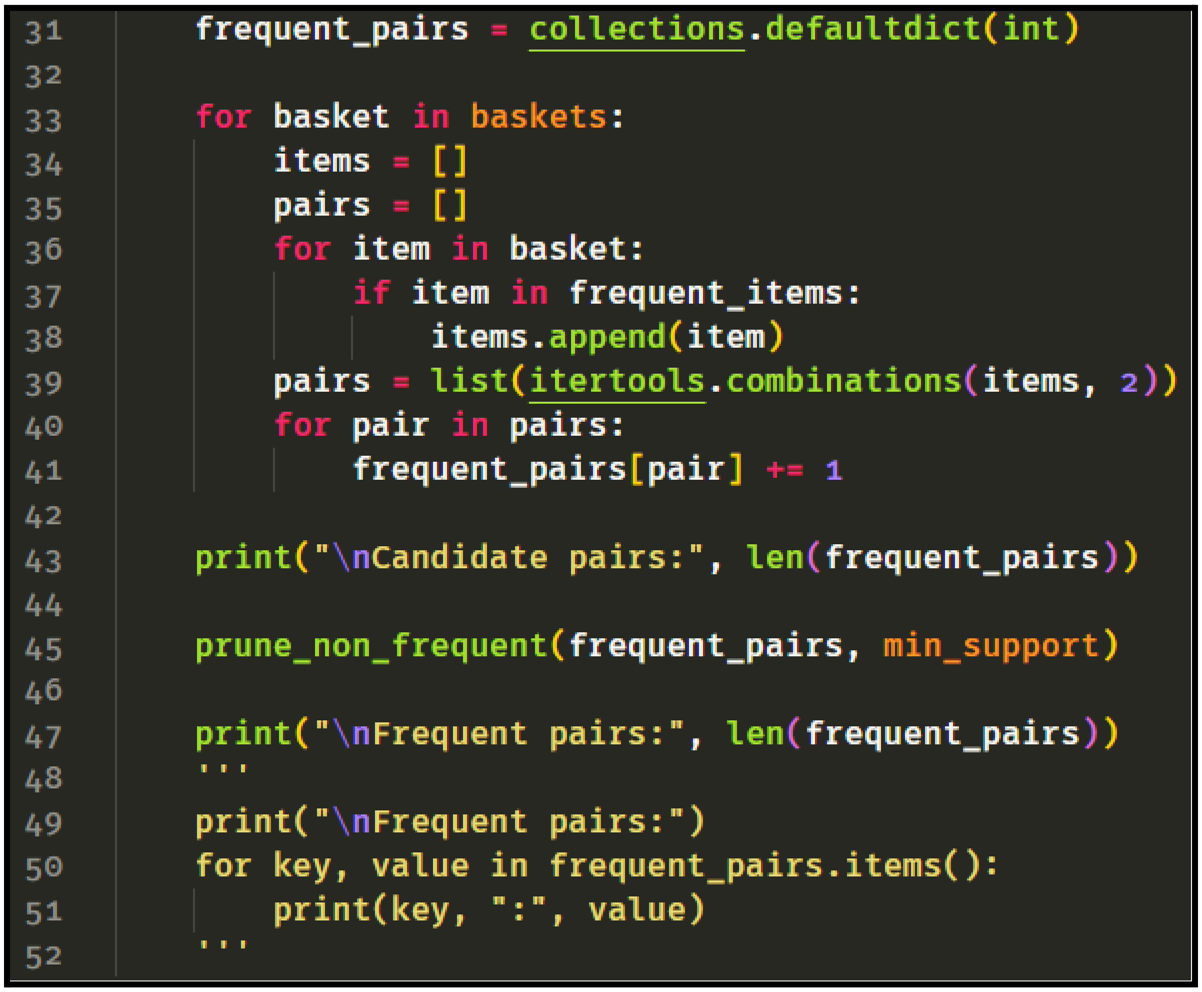
The data is read to the program by a 2D array, "baskets" is an array stored in every basket in the file. Each "basket" is an array stored in the elements. From line 15 to 19 are counting the occurrences of all items in the dataset**.** Line 23 is where removing all items that appear less times than the support number (s).



```
def prune_non_frequent(set_k, min_support):
    for key, value in list(set_k.items()):
        if value < min_support:
            del set_k[key]
```

This function is used to remove all infrequent items or pairs from the data.

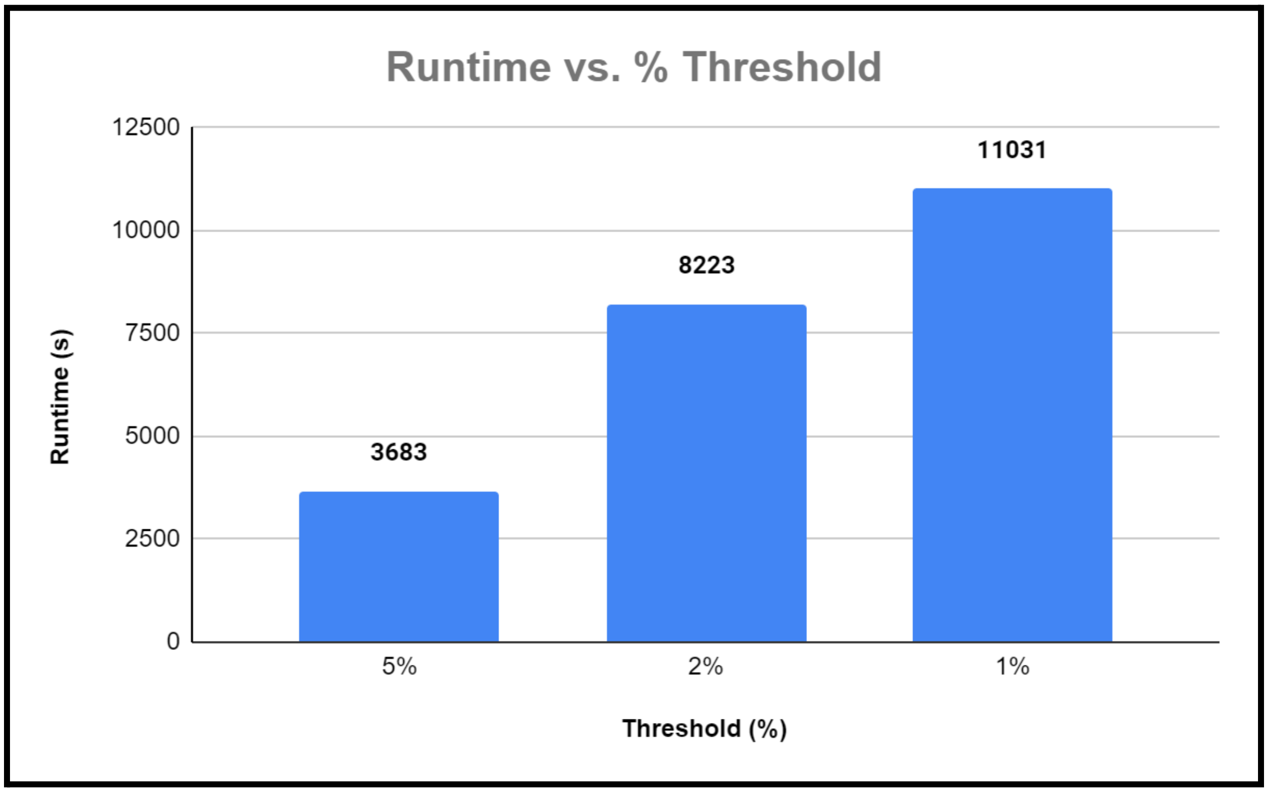
Pass 2: Read the baskets again and count only pairs that are constructed by both frequent items from pass 1. Prune all pairs that appear less times than the support number (s). The pairs passed the support threshold are the frequent pairs.



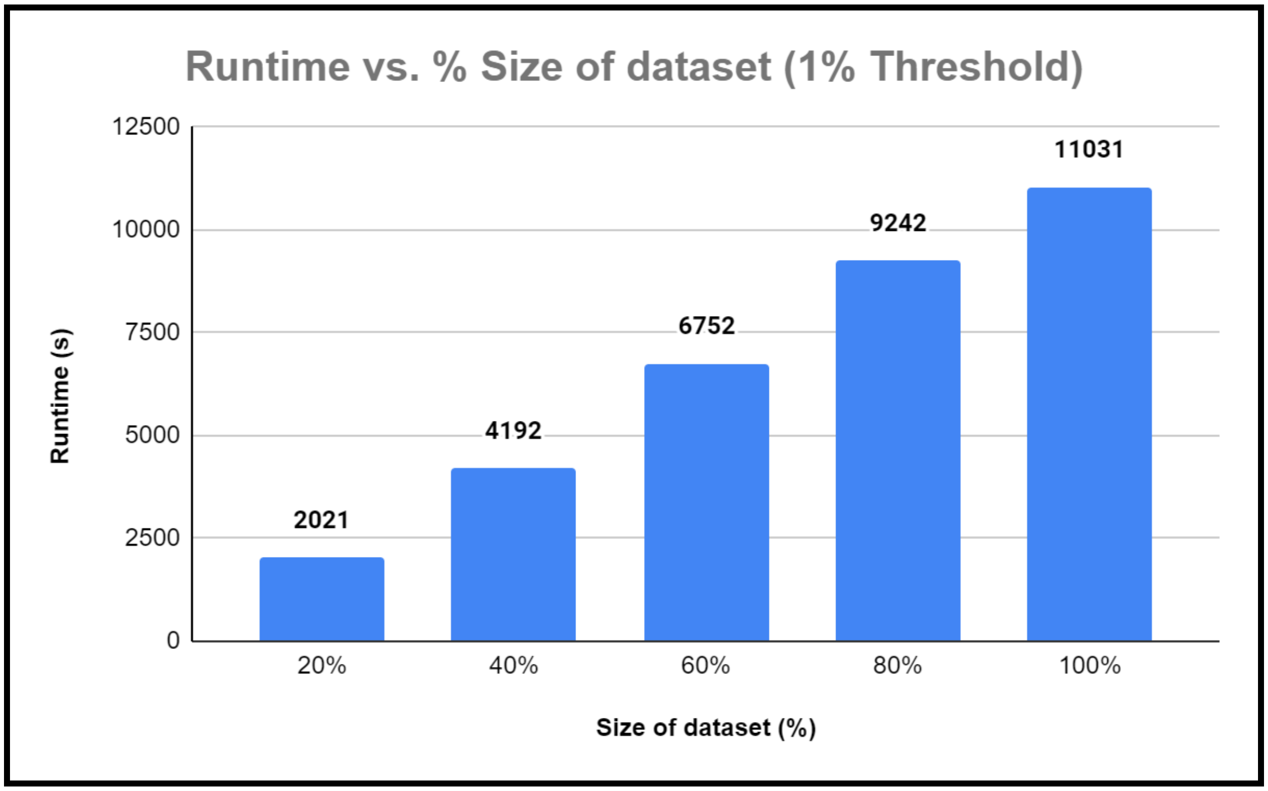
```
    frequent_pairs = collections.defaultdict(int)

    for basket in baskets:
        items = []
        pairs = []
        for item in basket:
            if item in frequent_items:
                items.append(item)
        pairs = list(itertools.combinations(items, 2))
        for pair in pairs:
            frequent_pairs[pair] += 1

    print("\nCandidate pairs:", len(frequent_pairs))

    prune_non_frequent(frequent_pairs, min_support)

    print("\nFrequent pairs:", len(frequent_pairs))
    '''
    print("\nFrequent pairs:")
    for key, value in frequent_pairs.items():
        print(key, ":", value)
    '''
```

For each basket in the data, the program will go through every items, if the item is frequent in pass 1, then add that item into a temporary list, after adding all frequent items in the basket, a python-built-in ("itertools.combinations") function will be used to make unique pairs from those items. And then counting those pairs into the candidate pairs list. After getting the candidate list, "prune_non_frequent" as used in pass 1, will remove all infrequent pairs from the candidate list. The leftover pairs are the frequent pairs.

- **Analysis:**



We can see that as threshold decreases, more frequent items (more than 1000 items added as threshold decreases), so more pairs are created, therefore, runtime increases.

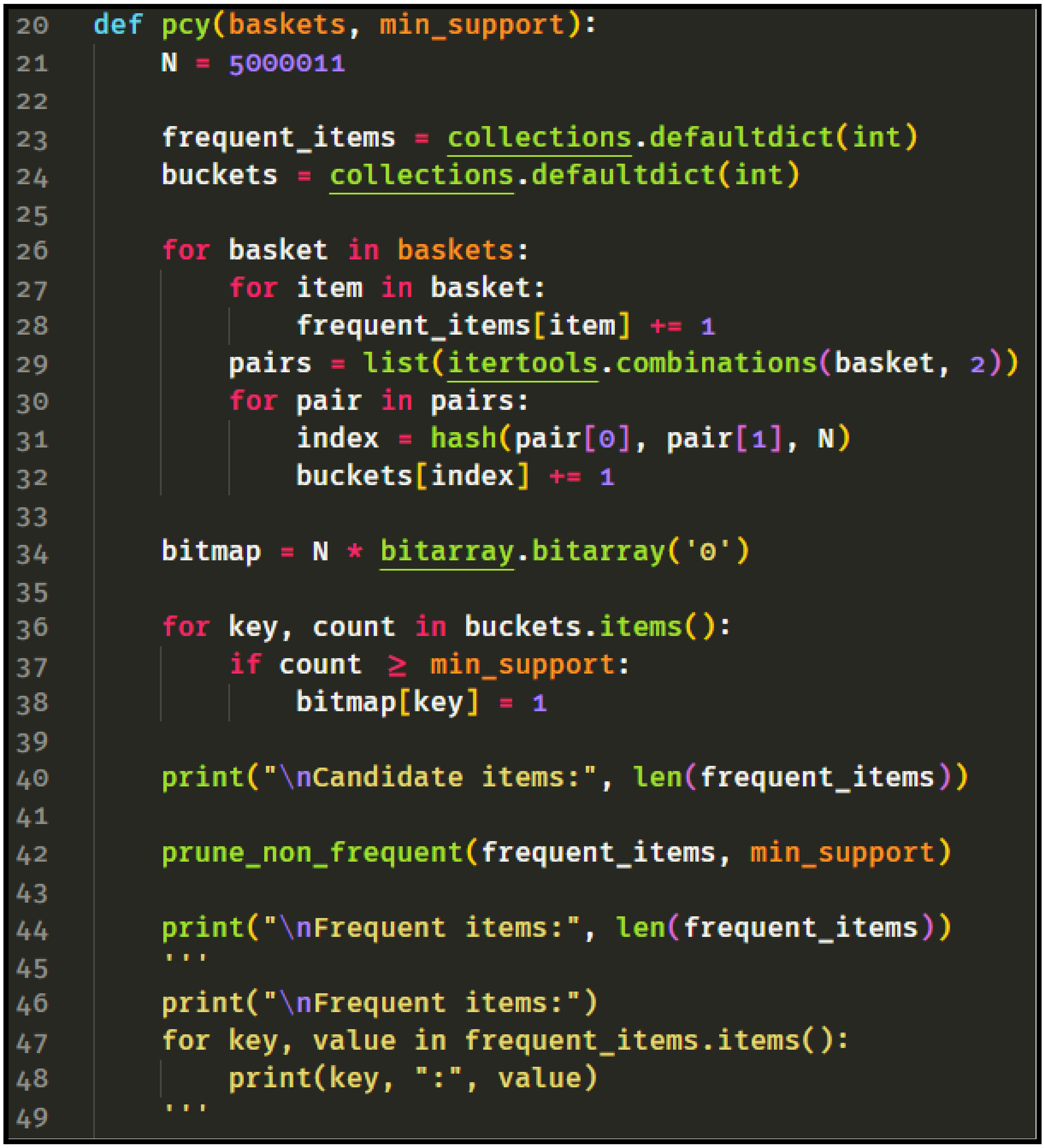


Runtime also increases as the size of the dataset increases. Since bigger datasets also means more items and pairs, therefore runtime increases.

PCY:

- **Code breakdown:**

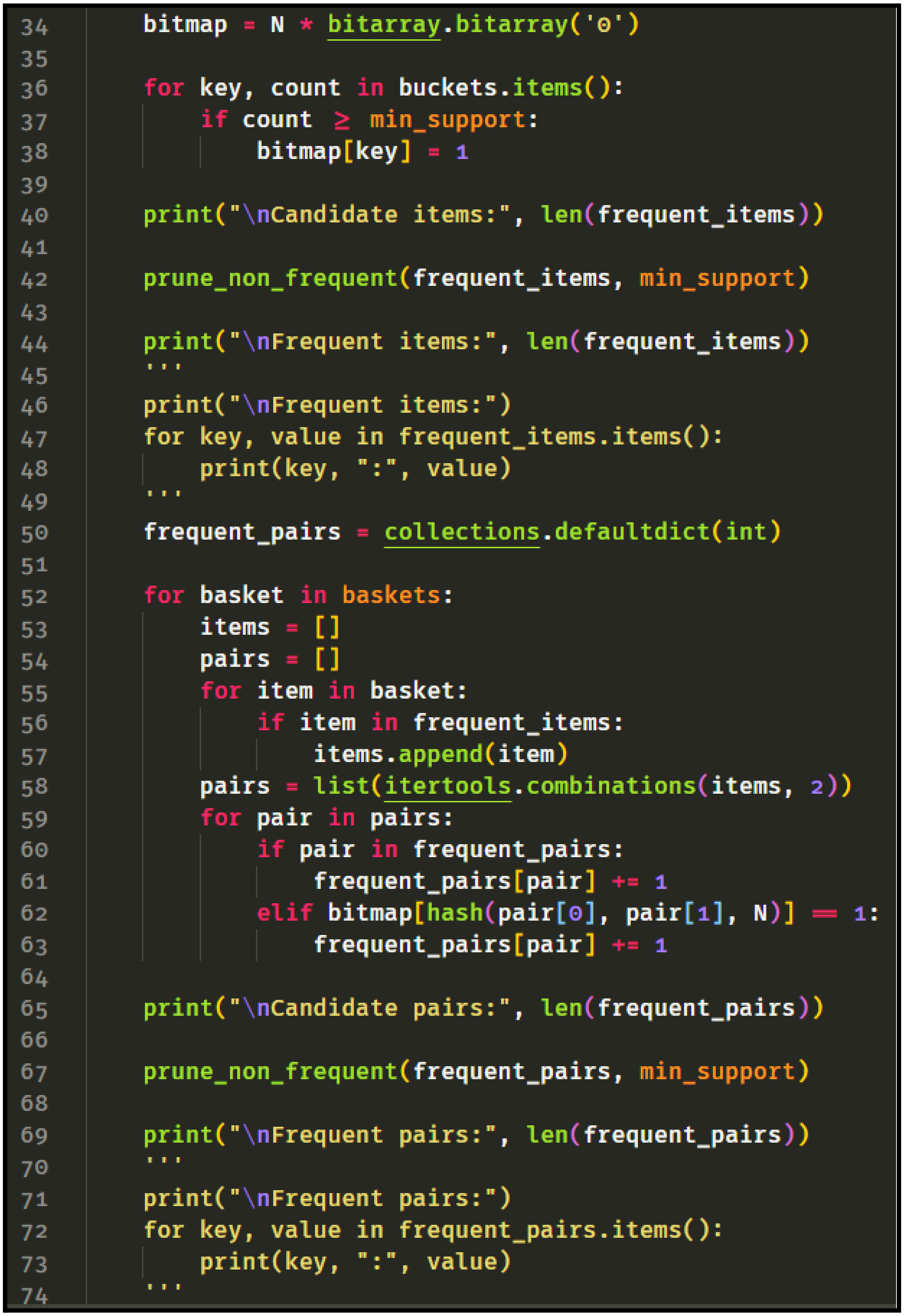
Pass 1:



```
20  def pcy(baskets, min_support):
21      N = 5000011
22  
23      frequent_items = collections.defaultdict(int)
24      buckets = collections.defaultdict(int)
25  
26      for basket in baskets:
27          for item in basket:
28              frequent_items[item] += 1
29          pairs = list(itertools.combinations(basket, 2))
30          for pair in pairs:
31              index = hash(pair[0], pair[1], N)
32              buckets[index] += 1
33  
34      bitmap = N * bitarray.bitarray('0')
35  
36      for key, count in buckets.items():
37          if count ≥ min_support:
38              bitmap[key] = 1
39  
40      print("\nCandidate items:", len(frequent_items))
41  
42      prune_non_frequent(frequent_items, min_support)
43  
44      print("\nFrequent items:", len(frequent_items))
45      '''
46      print("\nFrequent items:")
47      for key, value in frequent_items.items():
48          print(key, ":", value)
49      '''
```

Pass 1 of PCY is similar to Apriori, but with extra steps. After counting all items in the dataset, the program will create every possible pair, and hash them with the counts into a hash table; each bucket in the hash table might come from multiple pairs and counts (from line 26 to 32).

Pass 2:

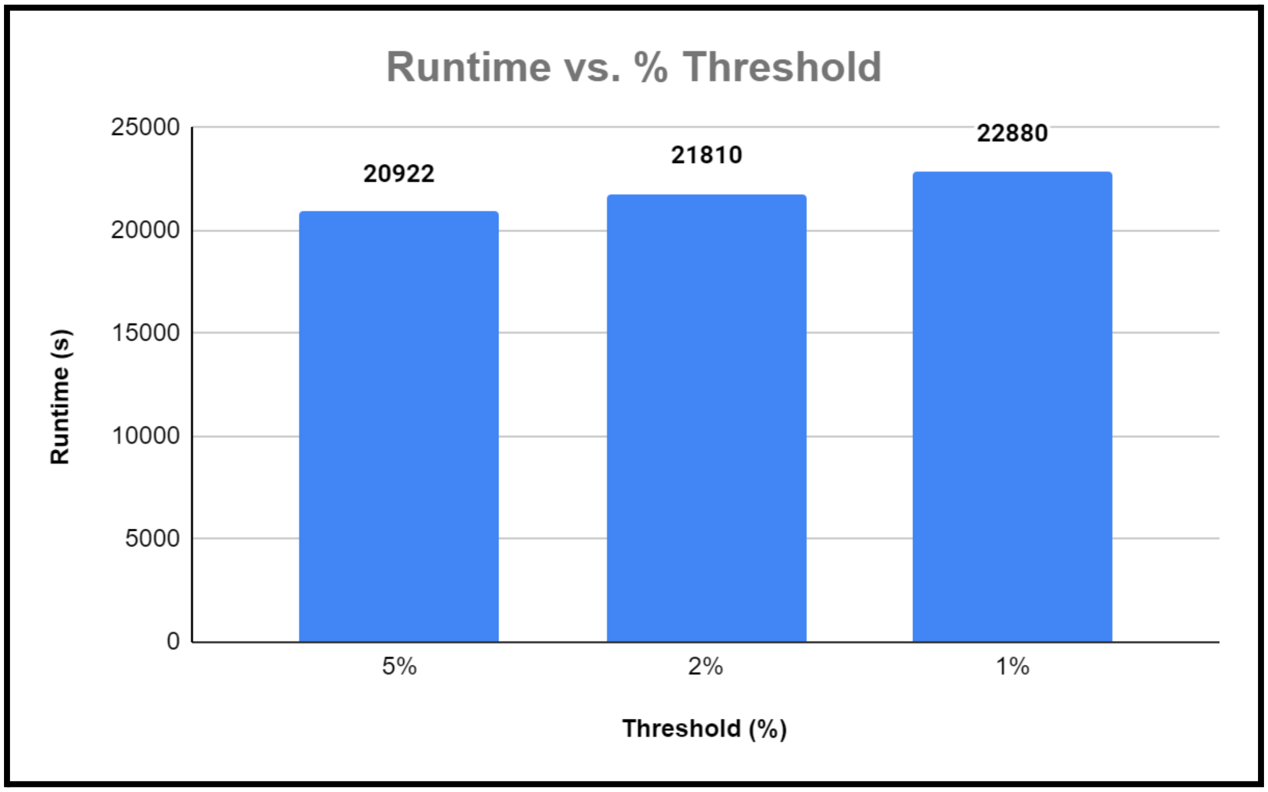


```python
    bitmap = N * bitarray.bitarray('0')

    for key, count in buckets.items():
        if count >= min_support:
            bitmap[key] = 1

    print("\nCandidate items:", len(frequent_items))

    prune_non_frequent(frequent_items, min_support)

    print("\nFrequent items:", len(frequent_items))
    '''
    print("\nFrequent items:")
    for key, value in frequent_items.items():
        print(key, ":", value)
    '''
    frequent_pairs = collections.defaultdict(int)

    for basket in baskets:
        items = []
        pairs = []
        for item in basket:
            if item in frequent_items:
                items.append(item)
        pairs = list(itertools.combinations(items, 2))
        for pair in pairs:
            if pair in frequent_pairs:
                frequent_pairs[pair] += 1
            elif bitmap[hash(pair[0], pair[1], N)] == 1:
                frequent_pairs[pair] += 1

    print("\nCandidate pairs:", len(frequent_pairs))

    prune_non_frequent(frequent_pairs, min_support)

    print("\nFrequent pairs:", len(frequent_pairs))
    '''
    print("\nFrequent pairs:")
    for key, value in frequent_pairs.items():
        print(key, ":", value)
    '''
```

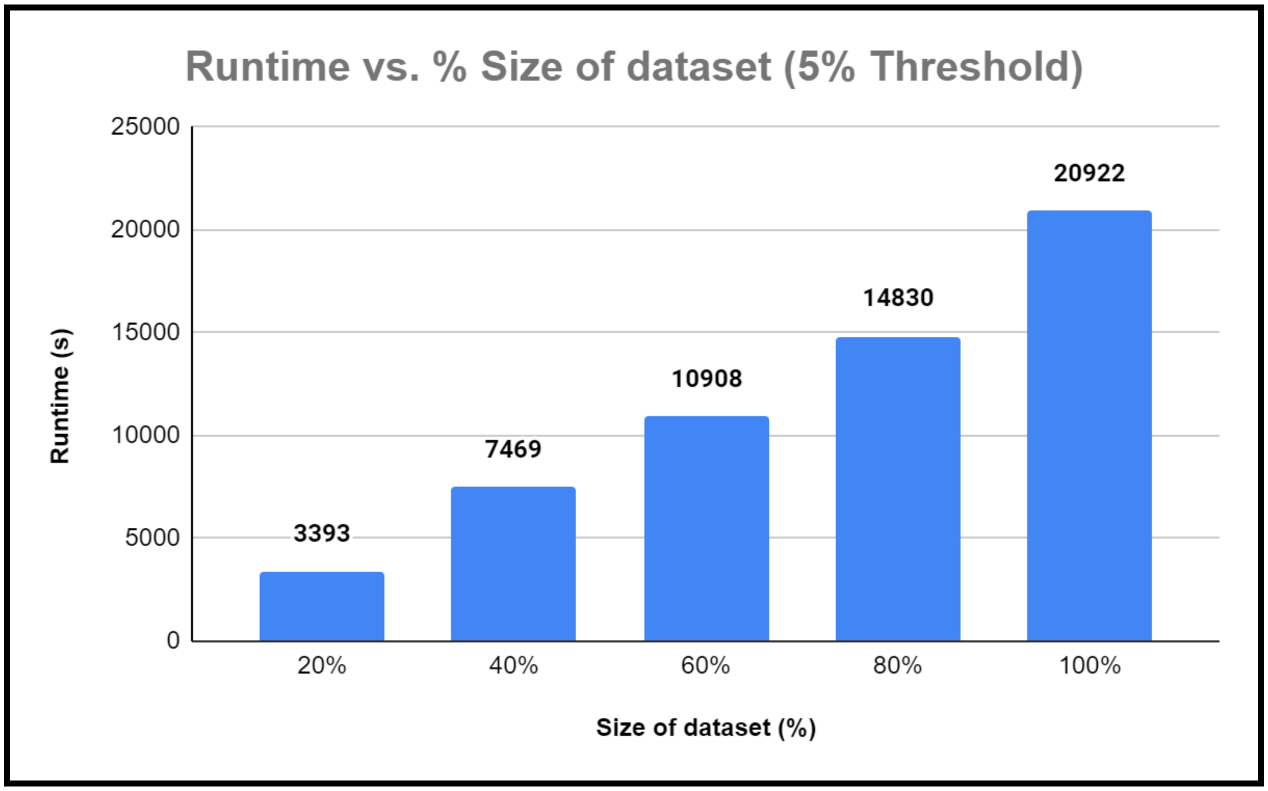
For pass 2, since, the program will create a bitmap to only count pairs in buckets that are frequent (buckets with count larger than % threshold). If a frequent pair is hashed into a bucket, that bucket must be frequent, but a frequent bucket doesn't mean all pairs hashed into that bucket are frequent. Therefore the program will check again each pair hashed to frequent buckets whether it is constructed by both frequent items, if it does, then add that pair to candidate pairs list and make counts for those pairs. After creating candidate pairs, remove all pairs that don't pass the threshold as in Apriori. The leftover pairs are frequent pairs.

Hash function and number of buckets is also important for PCY, a good hash function and good amount of buckets can decrease the amount of collisions in the hash table, therefore reducing more infrequent pairs in the candidate list. This program uses the Cantor pairing function link. The optimal number of buckets can vary by the amount of possible pairs the dataset has.

- **Analysis:**



Same pattern as Apriori, although the difference in runtime between thresholds are not as large as Apriori.



Same pattern as Apriori, runtime increases as the size of the dataset increases. Since bigger datasets also means more items and pairs, therefore runtime increases.

<u>Random Sampling & SON:</u>

- ### Random Sampling (RS):
  The difference between Random Sampling and Apriori is that RS randomly chooses a small percentage of the dataset to run Apriori on, whereas original Apriori runs on the hold data. Frequent items in the sample are frequent items in the hold data. Although, the program will need to verify all frequent pairs from sample data again with the hold data to eliminate false positives. Also, lower the threshold from the sample to find more pairs that are originally infrequent in the sample, but frequent in the hold data, in order to eliminate false negatives.

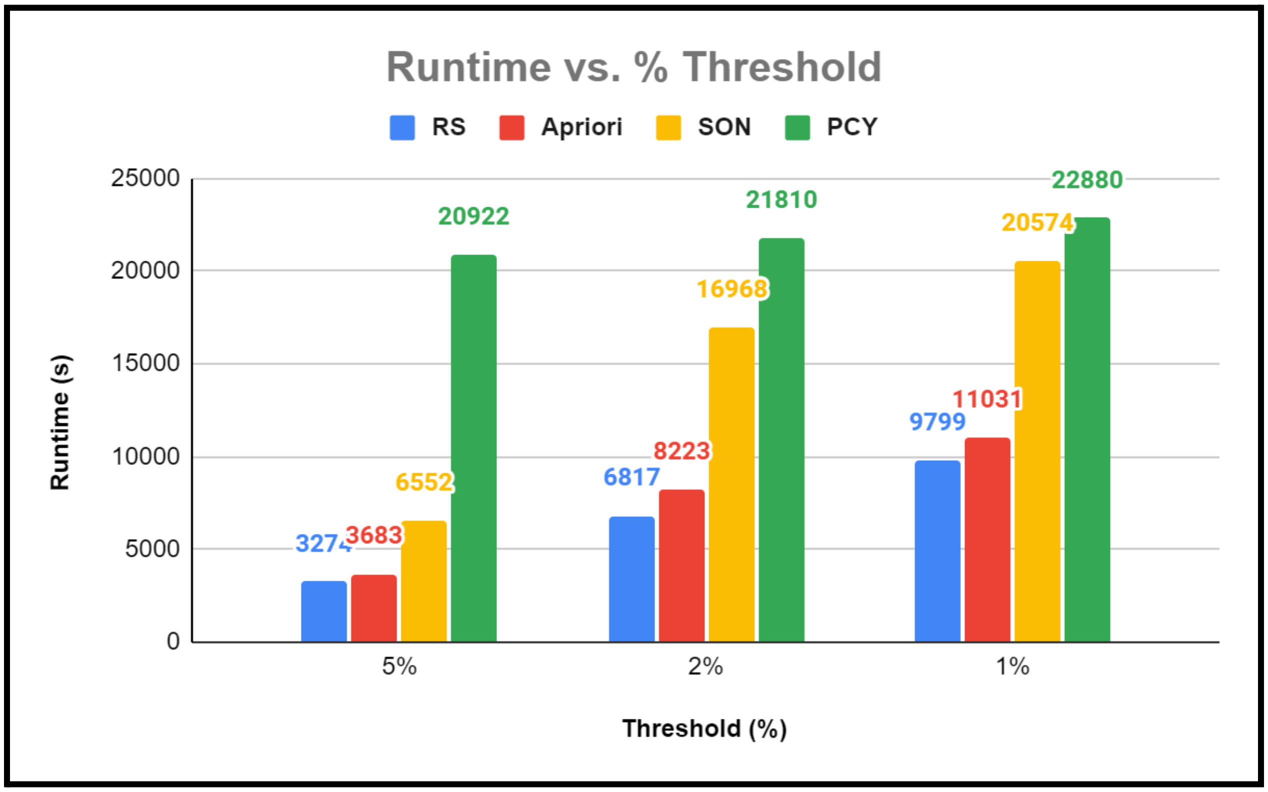
  The ideal sampling % for RS is less or equal to 10% of the data, because based on statistics, there is no difference between choosing samples larger than 10% of the population with samples less than 10%. Therefore, 10% would be the ideal sample size to maximize accuracy. In addition, adding an extra step in the RS algorithm to verify the frequent pairs from the sample with the hold data, by counting them again and verifying with the threshold of the hold data.

- ### SON:
  The difference between SON and Apriori is that SON splits the initial data into chunks and runs Apriori on each of them. SON is created for distributed computing, where one computer cannot run the hold the data at once, so it will split the data to multiple computers to run each of the chunks at the same time and combine the result at the end. Frequent items in the hold data need to be frequent in at least 1 of the chunks, but this

will create a lot of false positives, therefore the program will verify those frequent pairs from the chunks again with the hold data to eliminate false positives.

The ideal chunk size should be between 20% or 25% of the hold data for each chunk. Chunks that are too small will create a lot of false positives, whereas too large chunks don't utilize the distributed computing aspect of SON.

Apriori - PCY - Random Sampling - SON:



- RS is the best in terms of runtime, a bit less accurate than Apriori (false negatives).
- Apriori is the second best, balance between accuracy and performance.
- SON is only good with distributed computing, multi-threading.
- PCY is the slowest, memory efficient, scale better with different thresholds.

Multistage or multihash

- ### Multistage PCY:

For multistage PCY, the difference is it adds another pass in the middle with an extra hash function and table for all frequent pairs from the first hash tables to reduce more false positives. Frequent pairs need to be constructed by both frequent items and also hashed into both frequent buckets in the 2 hash tables. Hash functions need to be independent of each other.

- ### Multihash PCY:

Same concept with multistage but only 2 passes, the first pass has 2 hash tables instead of 1 as in original PCY. Frequent pairs need to be constructed by both frequent items and also hashed into both frequent buckets in the 2 hash tables. Hash functions also need to be independent of each other.