Q2 Part 1:

In this assignment, an A-priori approach is used compared to other implementation such as PCY or multi-hash. This is due to the use of hash functions and reallocation of memory for a bit map and count table after the values has been hashed for an optimised Bloom filter approach.

Pseudocode:

```
Sample dataset based on the given sample size percentage
Randomized the lines in each dataset
For each line:
	Initialise a list of baskets and bucket to record each unique item of the pairs
	Using the threshold of 1% to generate candidate sets
	Search for infrequent subsets among candidates
		If true, prune the itemset, else maintain the candidate set
	Obtain all the candidate item sets from the dataset assigned based on their count
	Allocate frequent items from the candidate item sets that meets the threshold of 1%
Return frequent sets
```

Q2 Part 2:

The SON algorithm is implemented based on the following pseudocode below. Recalling the algorithm divides baskets into partitions, it first mapped the local frequent itemset in the first pass to create a list of candidates and proceeded to implement A-priori again to determine the final frequent set before output. We then saved all the itemset based on the frequency in single and pairs and sort them into ascending order.

Pseudocode:

```
Sample dataset based on the given sample size percentage
Randomized the lines in each dataset
Allocate a parallel, distributed setting like Hadoop or Spark for better memory optimization
when data is allocated to simulate map-reduce operation
For each line:
	In the first pass
		Split dataset into chunks
		Map the baskets and bucket to record each unique item of the pairs
		Using the threshold of 1% to generate candidate sets
		Search for infrequent subsets among candidates
			If true, prune the itemset, else maintain the candidate set
		Obtain all candidate item sets from the dataset assigned based on their count
		Allocate frequent items from the candidate item sets that meets 1% threshold
	In second pass
		Repeat first pass
Return frequent sets
```

```
{'3', '4'}):250
{'8', '2'}):250
{'7', '4'}):250
{'1', '2'}):250
{'1', '4'}):250
{'6', '7'}):250
{'7', '2'}):250
{'9', '0'}):250
```

Diagram 1: An example of the output at chess.dat.txt with 5% sample size. The frequent itemset are identified with the count on the right

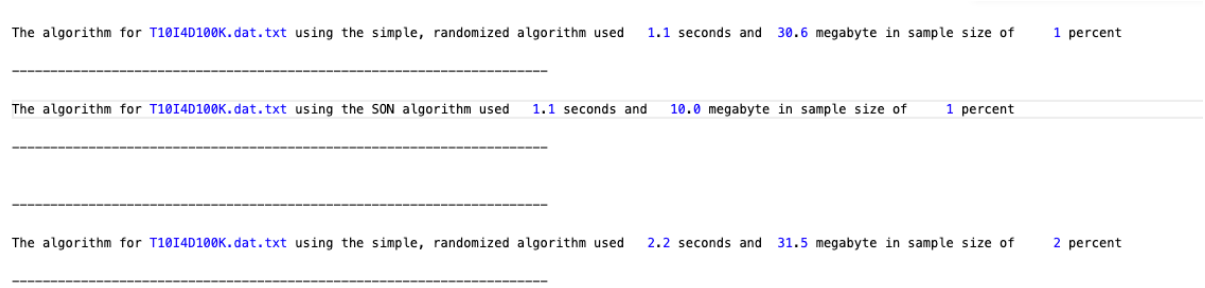


Diagram 2: An example of the log file

The following outputs and respective logbook can be found in the output directory attached in the zip file.

Comparison in time and memory usage on each dataset:
Through closer inspection of the log file, as sample size increases, the computational time and memory increases gradually for both randomized and SON algorithms. SON has a lower computational time and memory compared to the other in general due to the use of distributed data allocation through the first and second passes in SON. Upon observation in the difference in sample sizes, the rate of selecting a frequent item increase as the sample sizes incremented from the range 1%, 2%, 5% and 10%. This provides the algorithm with a higher likelihood of generating more accurate results as the probability of the minimal support threshold increases due to the increased sampling.

Challenges faced in this assignment while implementing the following algorithms would be the use of excessive computational time while passing through large datasets as the sample sizes increases. The potential of obtaining both false positives and negatives due to the inaccurate representation of the sample data collected may also lead to inaccuracy in frequent itemset generation in randomized algorithm. SON algorithm in this implementation minimizes it by approximately 5.3% in seconds in the case of the pumsb dataset, however such program leaves possible room of improvement if the sample size, minimal support threshold and confidence are well-selected.

Another consideration included memory consumptions and identifying optimal ways to reduce time while working on the program. Numerous attempts, including PCY and Map Reduce has been implemented, but the outcome has been unsatisfactory upon evaluation with the frequent pairs taken from another Python library in A-priori built in function within Python.

Q3 Part 1:
Greedy Algorithm is implemented by making their decision based on the response of towards each input element and the past. For instance, while assigning advertisement, given that a search query depends on the advertiser and its assigned budget to spare, an implementation of the greedy algorithm would be assigning the query to the highest bidder among the advertisers that consists of remaining budgets on hand. Once the advertiser no longer has any budgets left, no more queries will be assigned to the current advertiser and the next queries will be assigned to the subsequent highest bidder based on their budgets. This, however, means that unlike off-line algorithms, in a situation where multiple queries of

the same number are allocated, the greedy algorithm will assign the first query directly to the highest bidder, leaving no advertisement on the following queries. Therefore, this may result in an inefficiency in assigning advertiser for profit and poorer utilisation of the available advertisers on the given queries.

Balance Algorithm, which provides a competitive ratio of ¾, serves as an improvement for Greedy Algorithm by assigning a query to the bided advertiser in accordance with those that possessed the largest remaining budget, therefore breaking the ties subjectively. For example, assume that we have advertisers A and B having budgets of 2, followed by a set of queries xxyy, with A only bidding for x while B bids for both. In Balance Algorithm, we can allocate the two x's in both A and B in any given order as for the second x query, it will assign it to the advertiser that have the highest budget left. With a remaining budget of 1 in each advertiser in the revenue-optimised scenario, we can allocate B, which has the highest remaining budget left as a bidder, to the first y, leaving the last y to be unallocated. In an optimised condition where off-line algorithm is used to consider all advertiser, in this scenario we can compute the ratio as ¾, with a lower bound of (1 – 1/e) in ratio for a larger crowd of bidders.

For context, the competitive ratio is defined as the constant number of times at most where an online algorithm can assign queries in each range of advertisers in comparison to result from off-line algorithms. Expected to be less than 1, it may rely on the type of data input and the given restrictions of the bidders and/or queries.

Part 2:
Q1: Based on the greedy algorithm, it will start by assigning to the highest bidder. Given that all advertiser has the same amount of budget, it will assign the search query to the highest valued advertiser that has the budget, which is advertiser C that allows queries x, y and z. Starting from query x, we can illustrate the procedure as below:

| Query | Remaining budget | Allocated advertiser |
|---|---|---|
| x | 1 | C |
| x | 0 | C |
| y | 1 | B |
| y | 0 | B |

In the worst-case scenario, since the only advertiser left, which is A, does not bid for query z, the algorithm reaches its base case, despite A having remaining budgets. Hence the algorithm can assign at least four advertisers as shown in the table, given that the competitive ratio is 4/6 = 2/3.

Q2: One such sequence would be **xyxzzy**, where greedy algorithm can only assign 3 queries to the advertiser C, C and B in the respective order unlike 6 queries for offline algorithm, which will give us ½ competitive ratio.