12. Implement a Java program to perform Apriori algorithm

```
import java.util.*;

public class AprioriAlgorithm {

   // Helper function to find frequent itemsets
   public static List<Set<String>> apriori(List<Set<String>> transactions, double
minSupport)
{
        List<Set<String>> frequentItemsets = new ArrayList<>();
        Map<Set<String>, Integer> itemCountMap = new HashMap<>();

        // Single item sets (1-itemsets)
        for (Set<String> transaction : transactions) {
            for (String item : transaction) {
                Set<String> singleItemSet = new HashSet<>();
                singleItemSet.add(item);
                itemCountMap.put(singleItemSet, itemCountMap.getOrDefault(singleItemSet, 0) +
1);
            }
        }

        // Filter 1-itemsets by support
        double threshold = minSupport * transactions.size();
        itemCountMap.entrySet().removeIf(entry -> entry.getValue() < threshold);

        frequentItemsets.addAll(itemCountMap.keySet());

        // K-itemsets
        int k = 2;
        List<Set<String>> prevItemsets = new ArrayList<>(itemCountMap.keySet());
        while (!prevItemsets.isEmpty()) {
            Map<Set<String>, Integer> candidateItemCountMap = new HashMap<>();

            // Generate candidate itemsets (k-itemsets)
            for (int i = 0; i < prevItemsets.size(); i++) {
```

```java
            for (int j = i + 1; j < prevItemsets.size(); j++) {
                Set<String> candidate = new HashSet<>(prevItemsets.get(i));
                candidate.addAll(prevItemsets.get(j));
                if (candidate.size() == k) {
                    // Count occurrences in transactions
                    int count = 0;
                    for (Set<String> transaction : transactions) {
                        if (transaction.containsAll(candidate)) {
                            count++;
                        }
                    }
                    candidateItemCountMap.put(candidate, count);
                }
            }
        }

        // Filter by support
        candidateItemCountMap.entrySet().removeIf(entry -> entry.getValue() < threshold);
        frequentItemsets.addAll(candidateItemCountMap.keySet());
        prevItemsets = new ArrayList<>(candidateItemCountMap.keySet());
        k++;
    }
    return frequentItemsets;
}

public static void main(String[] args) {
    // Sample transactions
    List<Set<String>> transactions = Arrays.asList(
            new HashSet<>(Arrays.asList("Milk", "Bread", "Butter")),
            new HashSet<>(Arrays.asList("Bread", "Butter")),
            new HashSet<>(Arrays.asList("Milk", "Eggs")),
            new HashSet<>(Arrays.asList("Milk", "Bread", "Butter", "Eggs")),
            new HashSet<>(Arrays.asList("Bread", "Butter", "Cheese"))
    );
```

```java
        // Define minimum support (e.g., 0.6 means 60%)
        double minSupport = 0.6;

        // Call apriori algorithm
        List<Set<String>> frequentItemsets = apriori(transactions, minSupport);

        // Print the frequent itemsets
        System.out.println("Frequent Itemsets:");
        for (Set<String> itemset : frequentItemsets) {
            System.out.println(itemset);
        }
    }
}
```

OUTPUT:

```
Frequent Itemsets:
[Milk]
[Bread]
[Butter]
[Bread, Butter]
[Milk, Bread]
[Milk, Butter]
```

14. Write a program of cluster analysis using simple k-means algorithm Python programming language.

```python
import random
import numpy as np

# Function to calculate Euclidean distance between two points
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2) ** 2))

# Function to assign each point to the nearest centroid
```

```
def assign_clusters(points, centroids):
    clusters = {}
    for i in range(len(centroids)):
        clusters[i] = []

    for point in points:
        distances = [euclidean_distance(point, centroid) for centroid in centroids]
        closest_centroid = np.argmin(distances)
        clusters[closest_centroid].append(point)
    return clusters

# Function to update the centroids as the mean of the points in each cluster
def update_centroids(clusters):
    centroids = []
    for cluster in clusters.values():
        new_centroid = np.mean(cluster, axis=0)
        centroids.append(new_centroid)
    return centroids

# K-Means algorithm function
def k_means_clustering(points, k, max_iterations=100):
    # Randomly initialize centroids by choosing k random points
    centroids = random.sample(list(points), k)
    for _ in range(max_iterations):
        # Assign points to clusters based on the nearest centroid
        clusters = assign_clusters(points, centroids)

        # Update the centroids to be the mean of the points in each cluster
        new_centroids = update_centroids(clusters)

        # Check for convergence (if centroids don't change)
        if np.allclose(centroids, new_centroids):
            break
        centroids = new_centroids
    return centroids, clusters
```

```
# Main function
if __name__ == "__main__":
    # Sample 2D points
    points = np.array([
        [1.0, 1.0],
        [1.5, 2.0],
        [3.0, 4.0],
        [5.0, 7.0],
        [3.5, 5.0],
        [4.5, 5.0],
        [3.5, 4.5]
    ])

    # Number of clusters
    k = 2

    # Perform K-Means Clustering
    centroids, clusters = k_means_clustering(points, k)

    # Print the results
    print("Final centroids:", centroids)
    for cluster_idx, cluster_points in clusters.items():
        print(f"Cluster {cluster_idx + 1}: {cluster_points}")
```

OUTPUT

```
Final centroids: [array([1.25, 1.5 ]), array([3.9, 5.1])]
Cluster 1: [array([1., 1.]), array([1.5, 2. ])]
Cluster 2: [array([3., 4.]), array([5., 7.]), array([3.5, 5. ]), array([4.5, 5. ]), array([3.5, 4.5])]
```