Scala Programs

1.

Write the scala code to implement bubble sort algorithm.

```
object BubbleSort{
  def bubblesort(arr:Array[Int]): Unit={
    val n=arr.length
    var swapped=true

    while(swapped==true) {
      swapped=false
      for(i <- 1 until n){
        if(arr(i-1)>arr(i)){
          val temp= arr(i)
          arr(i)= arr(i-1)
          arr(i-1)=temp
          swapped=true
        }
      }
    }
  }

  def main(args:Array[String]):Unit={
    val arr=Array(12,34,45,242,12,98,2)
    println("Original Array:")
    arr.foreach(println)
```

```
    bubblesort(arr)
    println("\nSorted Array:")
    arr.foreach(println)
  }
}
```

2.

Write scala code to find the length of each word from the array.

```
object WordLengths {
 def main(args: Array[String]): Unit = {
  val words = Array("apple", "banana", "cherry", "date", "elderberry", "fig", "grape")
  val wordLengths = words.map(word => (word, word.length))

  println("Word lengths:")
  wordLengths.foreach { case (word, length) => println(s"$word: $length") }
 }
}
```

3.

Write scala code to find the number of books published by each author, referring to the collection given below, with (authorName, BookName)

{ (‘ Dr. Seuss’: ‘How the Grinch Stole Christmas!’) ,(‘ Jon Stone’: ‘Monsters at the End of This Book’) , (‘ Dr. Seuss’: ‘The Lorax’) ,(‘Jon Stone’: ‘Big Bird in China’) (‘ Dr. Seuss’ :’ One Fish, Two Fish, Red Fish, Blue Fish’) }

CODE:

```
object BookCountByAuthor {
  def main(args: Array[String]): Unit = {
    // Given collection of tuples (authorName, bookName)
    val books = List(
      ("Dr. Seuss", "How the Grinch Stole Christmas!"),
      ("Jon Stone", "Monsters at the End of This Book"),
      ("Dr. Seuss", "The Lorax"),
      ("Jon Stone", "Big Bird in China"),
      ("Dr. Seuss", "One Fish, Two Fish, Red Fish, Blue Fish")
    )

    val bookCount = books.groupBy(_._1).view.mapValues(_.size).toMap

    bookCount.foreach { case (author, count) =>
      println(s"$author published $count book(s)")
    }
  }
}
```

4. Write the program to illustrate the use of pattern matching in scala, for the following

Matching on case classes.

Define two case classes as below:

abstract class Notification

case class Email(sender: String, title: String, body: String) extends Notification

case class SMS(caller: String, message: String) extends Notification

Define a function showNotification which takes as a parameter the abstract type Notification and matches on the type of Notification (i.e. it figures out whether it's an Email or SMS).

In the case it’s an Email(email, title, _) return the string: s"You got an email from $email with title: $title“

In the case it’s an SMS return the String: s"You got an SMS from $number! Message: $message“

CODE:

```
abstract class Notification


case class Email(sender: String, title: String, body: String) extends Notification

case class SMS(caller: String, message: String) extends Notification


object NotificationExample {
  def showNotification(notification: Notification): String = {
    notification match {
      case Email(sender, title, _) =>
        s"You got an email from $sender with title: $title"
      case SMS(caller, message) =>
        s"You got an SMS from $caller! Message: $message"
    }
  }


  def main(args: Array[String]): Unit = {
    val email = Email("john.doe@example.com", "Meeting Reminder", "Don't forget the meeting at 10 AM.")
    val sms = SMS("123-456-7890", "Your package has been delivered.")


    println(showNotification(email))
    println(showNotification(sms))
  }
}
```

5. Write the scala program using imperative style to implement quick sort algorithm.

CODE:

```
object QuickSort {
  def quickSort(arr: Array[Int]): Unit = {
    def swap(i: Int, j: Int): Unit = {
      val temp = arr(i)
      arr(i) = arr(j)
      arr(j) = temp
    }

    def partition(low: Int, high: Int): Int = {
      val pivot = arr(high)
      var i = low - 1
      for (j <- low until high) {
        if (arr(j) <= pivot) {
          i += 1
          swap(i, j)
        }
      }
      swap(i + 1, high)
      i + 1
    }

    def quickSortRecursive(low: Int, high: Int): Unit = {
      if (low < high) {
        val pi = partition(low, high)
```

```
      quickSortRecursive(low, pi - 1)
      quickSortRecursive(pi + 1, high)
    }
  }

  quickSortRecursive(0, arr.length - 1)
 }

 def main(args: Array[String]): Unit = {
  val arr = Array(10, 7, 8, 9, 1, 5)
  println("Unsorted array: " + arr.mkString(", "))
  quickSort(arr)
  println("Sorted array: " + arr.mkString(", "))
 }
}
```

6. Write a scala function to convert the each word to capitalize each word in the given sentence.

CODE:

```
object CapitalizeWords {
 def capitalizeEachWord(sentence: String): String = {
  sentence.split(" ").map(word => word.capitalize).mkString(" ")
 }

 def main(args: Array[String]): Unit = {
  val sentence = "hello world this is scala"
  val capitalizedSentence = capitalizeEachWord(sentence)
  println(capitalizedSentence) // Output: "Hello World This Is Scala"
 }
```

}

7. Write scala code to show functional style program to implement quick sort algorithm.

8. For the below given collection of items with item-names and quantity, write the scala code for the given statement.

Items = {(“Pen”:20), (“Pencil”:10), (“Erasor”:7), (“Book”:25), (“Sheet”:15)}

i. Display item-name and quantity

ii. Display sum of quantity and total number of items

iii. Add 3 Books to the collection

Add new item “Board” with quantity 15 to the collection

CODE:

```
object ItemCollection {
 def main(args: Array[String]): Unit = {
  // Define the initial collection of items as a Map
  var items = Map("Pen" -> 20, "Pencil" -> 10, "Eraser" -> 7, "Book" -> 25, "Sheet" -> 15)

  // i. Display item-name and quantity
  println("Items in the collection:")
  items.foreach { case (item, quantity) =>
   println(s"$item : $quantity")
  }
```

```
  // ii. Display sum of quantity and total number of items
  val totalQuantity = items.values.sum
  val totalItems = items.size
  println(s"\nTotal Quantity: $totalQuantity")
  println(s"Total Number of Items: $totalItems")

  // iii. Add 3 Books to the collection
  items += ("Book" -> (items.getOrElse("Book", 0) + 3))

  // Add new item "Board" with quantity 15 to the collection
  items += ("Board" -> 15)

  // Display the updated collection after additions
  println("\nUpdated Items in the collection after additions:")
  items.foreach { case (item, quantity) =>
    println(s"$item : $quantity")
  }
 }
}
```

9. Develop a scala code to search an element in the given list of numbers. The function Search() will take two arguments: list of numbers and the number to be searched. The function will write True if the number is found, False otherwise.

CODE:

```
object NumberSearch {
 def Search(numbers: List[Int], target: Int): Boolean = {
  numbers.contains(target)
```

```
  }

  def main(args: Array[String]): Unit = {
    val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    val target1 = 5
    val target2 = 15

    println(s"Is $target1 in the list? ${Search(numbers, target1)}")
    println(s"Is $target2 in the list? ${Search(numbers, target2)}")
  }
}
```

10. Illustrate the implementation by writing scala code to generate a down counter from 10 to 1.

CODE:

```
object DownCounterForLoop {
  def main(args: Array[String]): Unit = {
    // Down counter using for loop
    for (i <- 10 to 1 by -1) {
      println(i)
    }
  }
}
```

11. Design a scala function to perform factorial item in the given collection. The arguments passed to the function are the collection of items. Assume the type of the argument for the function suitably. The return type is to be integer.

CODE:

```
object FactorialCollection {
 // Function to calculate factorial of a number
 def factorial(n: Int): Int = {
  if (n == 0 || n == 1) 1
  else n * factorial(n - 1)
 }

 // Function to compute factorial for each item in the collection
 def computeFactorials(numbers: List[Int]): List[Int] = {
  numbers.map(factorial)
 }

 def main(args: Array[String]): Unit = {
  val numbers = List(5, 3, 7, 2, 4)

  // Compute factorials for the numbers in the list
  val factorialResults = computeFactorials(numbers)

  // Print the results
  factorialResults.zipWithIndex.foreach { case (factorialResult, index) =>
   println(s"Factorial of ${numbers(index)} is $factorialResult")
  }
 }
}
```

12. For the below given collection of items with item names and quantity, write the scala code for the given statement. Items = {(“Butter”:20), (“Bun”:10), (“Egg”:7), (“Biscuit”:25),(“Bread”:15)}

i. Display item-name and quantity

ii. Display sum of quantity and total number of items

iii. Add 3 Buns to the collection

iv. Add new item “Cheese” with quantity 12 to the collection

CODE:

```
object ItemOperations {
 def main(args: Array[String]): Unit = {
  // Given collection of items as a Map
  var items = Map("Butter" -> 20, "Bun" -> 10, "Egg" -> 7, "Biscuit" -> 25, "Bread" -> 15)

  // i. Display item-name and quantity
  println("Items in the collection:")
  items.foreach { case (item, quantity) =>
   println(s"$item : $quantity")
  }

  // ii. Display sum of quantity and total number of items
  val totalQuantity = items.values.sum
  val totalItems = items.size
  println(s"\nTotal Quantity: $totalQuantity")
  println(s"Total Number of Items: $totalItems")

  // iii. Add 3 Buns to the collection
  items += ("Bun" -> (items.getOrElse("Bun", 0) + 3))
```

```
  // iv. Add new item "Cheese" with quantity 12 to the collection
  items += ("Cheese" -> 12)

  // Display the updated collection after additions
  println("\nUpdated Items in the collection after additions:")
  items.foreach { case (item, quantity) =>
    println(s"$item : $quantity")
  }
 }
}
```

13.Implement function for binary search using recursion in Scala to find the number, given a list of numbers. The function will have two arguments: Sorted list of numbers and the number to be searched.

CODE:

```
object BinarySearch {
 def binarySearch(sortedList: List[Int], target: Int): Boolean = {
  def binarySearchHelper(left: Int, right: Int): Boolean = {
   if (left > right) false
   else {
    val mid = left + (right - left) / 2
    sortedList(mid) match {
     case `target` => true
     case x if x > target => binarySearchHelper(left, mid - 1)
     case _ => binarySearchHelper(mid + 1, right)
    }
   }
  }
```

```
    binarySearchHelper(0, sortedList.length - 1)
  }

  def main(args: Array[String]): Unit = {
    val numbers = List(2, 4, 6, 8, 10, 12, 14, 16)
    val target1 = 8
    val target2 = 5

    println(s"Searching for $target1 in the list: ${binarySearch(numbers, target1)}")
    println(s"Searching for $target2 in the list: ${binarySearch(numbers, target2)}")
  }
}
```

14.Write a function to find the length of each word and return the word with highest length .

Ex for the collection of words = (“games”, “television”,”rope”,”table”)

The function should return (“television”,10). The word with the highest length .

Read the words from the keyboard.

CODE:

```
import scala.io.StdIn

object HighestWordLength {
  def main(args: Array[String]): Unit = {
    println("Enter words separated by spaces:")
    val words = StdIn.readLine().trim().split("\\s+")
```

```
  if (words.isEmpty) {
    println("No words entered. Exiting.")
  } else {
    val maxLengthWord = words.maxByOption(_.length)

    maxLengthWord match {
      case Some(word) => println(s"The word with the highest length is '$word' with length ${word.length}.")
      case None => println("No valid words found.")
    }
  }
 }
}
```

Spark Programs

15. Analyze the application of fold() and aggregate() functions in Spark by considering a scenario where all the items in a collection are updated by a count of 100. Evaluate the efficiency, performance, and suitability of both.

CODE:

```
import org.apache.spark.sql.SparkSession

object FoldAggregateExample {
 def main(args: Array[String]): Unit = {
  val spark = SparkSession.builder()
    .appName("Fold vs Aggregate Example")
    .master("local[*]")
```

```
    .getOrCreate()

  val sc = spark.sparkContext
  val data = sc.parallelize(1 to 1000000)

  // Using fold
  val foldedResult = data.fold(0)((acc, value) => acc + value + 100)

  // Using aggregate
  val aggregatedResult = data.aggregate(0)(
    (acc, value) => acc + value + 100,
    (acc1, acc2) => acc1 + acc2
  )

  println(s"Fold result: $foldedResult")
  println(s"Aggregate result: $aggregatedResult")

  spark.stop()
 }
}
```

16. Consider a text file text.txt. Develop Spark code to read the file and count the number of occurrences of each word using Spark RDD. Store the result in a file. Display the words which appear more than 4 times.

CODE:

16

```
import org.apache.spark.{SparkConf, SparkContext}
```

```
object WordCount {
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("WordCount").setMaster("local")
val sc = new SparkContext(conf)

// Step 2: Read the text file into an RDD
val inputFile = "path/to/text.txt" // Replace with the path to your text file
val lines = sc.textFile(inputFile)

// Step 3: Perform the word count
val words = lines.flatMap(line => line.split("\\W+")) // Split on non-word characters
val wordCounts = words.map(word => (word.toLowerCase, 1)).reduceByKey(_ + _)

// Step 4: Filter words that appear more than 4 times
val frequentWords = wordCounts.filter { case (word, count) => count > 4 }

// Step 5: Save the result to a file
val outputFile = "path/to/output" // Replace with the desired output directory
frequentWords.saveAsTextFile(outputFile)

// Step 6: Collect and display the frequent words
val result = frequentWords.collect()
println("Words that appear more than 4 times:")
result.foreach { case (word, count) => println(s"$word: $count") }

// Stop the Spark context
sc.stop()
```

```
}
}
```

17. Consider the content of text file text.txt. Perform the counting of occurrences of each word using pair RDD.

CODE:

```
import org.apache.spark.sql.SparkSession

object WordCount {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .appName("Word Count")
      .master("local[*]")
      .getOrCreate()

    val sc = spark.sparkContext

    // Read the text file into an RDD
    val inputFile = "path/to/text.txt"
    val textFile = sc.textFile(inputFile)

    // Perform word count using pair RDDs
    val wordCounts = textFile
      .flatMap(line => line.split("\\s+")) // Split each line into words
      .filter(word => word.nonEmpty) // Filter out empty words
      .map(word => (word, 1)) // Map each word to a pair (word, 1)
```

```
    .reduceByKey(_ + _) // Reduce by key to count occurrences

  // Collect and print the results
  wordCounts.collect().foreach { case (word, count) =>
    println(s"$word: $count")
  }

  spark.stop()
 }
}
```

18. Write the Spark Code to print the top 10 tweeters.

Tweet Mining: A dataset with the 8198 reduced tweets, reduced-tweets.json will be provided. The data contains reduced tweets as in the sample below:

{"id":"572692378957430785",

"user":"Srkian_nishu smile",

"text":"@always_nidhi @YouTube no idnt understand bti loved of this mve is rocking",

"place":"Orissa",

"country":"India"}

A function to parse the tweets into an RDD will be provided.

CODE:

18

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

object TopTweeters {
```

```
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context and Spark session
val conf = new SparkConf().setAppName("TopTweeters").setMaster("local")
val sc = new SparkContext(conf)
val spark = SparkSession.builder().config(conf).getOrCreate()

// Step 2: Read the JSON file into a DataFrame
val inputFile = "path/to/reduced-tweets.json" // Replace with the path to your JSON file
val tweetsDF = spark.read.json(inputFile)

// Step 3: Create an RDD from the user column
val usersRDD = tweetsDF.select("user").rdd.map(row => row.getString(0))

// Step 4: Perform the word count (user count) using pair RDD operations
val userPairs = usersRDD.map(user => (user, 1))
val userCounts = userPairs.reduceByKey(_ + _)

// Step 5: Get the top 10 tweeters
val topTweeters = userCounts.sortBy(_._2, ascending = false).take(10)

// Step 6: Print the top 10 tweeters
println("Top 10 Tweeters:")
topTweeters.foreach { case (user, count) => println(s"$user: $count") }

// Stop the Spark context
sc.stop()
spark.stop()
}
}
```

19. Simulate the following scenario using Spark streaming. There will be a process which will be streaming lines of text to a unix port using socket communication. The process we can use for this purpose is netcat. It will stream lines typed on the console to a unix socket. The spark application needs to read the lines from the specified port, and it needs to produce the word counts on the console. A batch interval of 5 second can be used.

CODE:

19

```
import org.apache.spark.SparkConf

import org.apache.spark.streaming.{Seconds, StreamingContext}

import org.apache.spark.streaming.dstream.DStream


object SocketWordCount {

def main(args: Array[String]): Unit = {

// Step 1: Create a local StreamingContext with two working threads and batch interval of 5 seconds

val conf = new SparkConf().setAppName("SocketWordCount").setMaster("local[2]")

val ssc = new StreamingContext(conf, Seconds(5))


// Step 2: Create a DStream that will connect to hostname:port, like localhost:9999

val lines = ssc.socketTextStream("localhost", 9999)


// Step 3: Split each line into words

val words: DStream[String] = lines.flatMap(_.split("\\W+"))


// Step 4: Count each word in each batch

val pairs: DStream[(String, Int)] = words.map(word => (word, 1))

val wordCounts: DStream[(String, Int)] = pairs.reduceByKey(_ + _)
```

```
// Step 5: Print the word counts
wordCounts.print()

// Step 6: Start the computation
ssc.start()

// Step 7: Wait for the computation to terminate
ssc.awaitTermination()
}
}
```

20. Develop the spark code to find the average of marks using the combineByKey() operation.

Sample Input format: Array(("Joe", "Maths", 83), ("Joe", "Physics", 74), ("Joe", "Chemistry", 91), ("Joe", "Biology", 82), ("Nik", "Maths", 69), ("Nik ", "Physics", 62), ("Nik ", "Chemistry", 97), ("Nik ", "Biology", 80))

CODE:

20

```
import org.apache.spark.{SparkConf, SparkContext}

object AverageMarks {
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("AverageMarks").setMaster("local")
val sc = new SparkContext(conf)
```

```
// Sample Input Data
val data = Array(
("Joe", "Maths", 83), ("Joe", "Physics", 74), ("Joe", "Chemistry", 91), ("Joe", "Biology", 82),
("Nik", "Maths", 69), ("Nik", "Physics", 62), ("Nik", "Chemistry", 97), ("Nik", "Biology", 80)
)

// Step 2: Create an RDD from the sample data
val rdd = sc.parallelize(data)

// Step 3: Create a pair RDD with (student, marks)
val studentMarks = rdd.map { case (student, subject, marks) => (student, marks) }

// Step 4: Use combineByKey to compute sum of marks and count of subjects for each student
val combineByKeyResult = studentMarks.combineByKey(
(marks: Int) => (marks, 1), // CreateCombiner: initial (sum, count)
(acc: (Int, Int), marks: Int) => (acc._1 + marks, acc._2 + 1), // MergeValue: update (sum, count)
(acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2) // MergeCombiners: combine (sum, count)
)

// Step 5: Calculate the average marks for each student
val averageMarks = combineByKeyResult.map { case (student, (sum, count)) => (student, sum.toDouble / count) }

// Step 6: Collect and print the results
val result = averageMarks.collect()
println("Average marks for each student:")
result.foreach { case (student, avgMarks) => println(s"$student: $avgMarks") }
```

```
// Stop the Spark context
sc.stop()
}
}
```

21. Consider the Employee table with the fields as (EmpID, Dept, EmpDesg). Design the Spark program to partition the table using Dept and construct the hashed partition of 4.

CODE:

21

```
import org.apache.spark.{SparkConf, SparkContext, Partitioner}
import org.apache.spark.rdd.RDD

object EmployeePartitionerExample {
case class Employee(EmpID: String, Dept: String, EmpDesg: String)

def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("EmployeePartitioner").setMaster("local")
val sc = new SparkContext(conf)

// Sample Input Data
val employeeData = Array(
Employee("1", "HR", "Manager"),
Employee("2", "Finance", "Analyst"),
Employee("3", "IT", "Developer"),
Employee("4", "IT", "Manager"),
Employee("5", "HR", "Executive"),
```

```
Employee("6", "Finance", "Manager"),
Employee("7", "IT", "Analyst"),
Employee("8", "HR", "Developer")
)

// Step 2: Create an RDD from the sample data
val employeeRDD: RDD[Employee] = sc.parallelize(employeeData)

// Step 3: Map the RDD to pair RDD with Dept as the key
val pairRDD = employeeRDD.map(emp => (emp.Dept, emp))

// Step 4: Define a custom partitioner
class DeptPartitioner(partitions: Int) extends Partitioner {
require(partitions > 0, s"Number of partitions ($partitions) must be positive.")
override def numPartitions: Int = partitions
override def getPartition(key: Any): Int = {
key.hashCode % numPartitions
}
}

// Step 5: Partition the RDD using the custom partitioner
val partitionedRDD = pairRDD.partitionBy(new DeptPartitioner(4))

// Step 6: Save the partitioned RDD to files (optional)
val outputDir = "path/to/output" // Replace with the desired output directory
partitionedRDD.saveAsTextFile(outputDir)

// Step 7: Verify the partitions (for demonstration purposes)
partitionedRDD.mapPartitionsWithIndex((index, iterator) => iterator.map((index, _)))
```

```
.collect()
.foreach { case (index, (dept, emp)) =>
println(s"Partition: $index, Dept: $dept, Employee: $emp")
}

// Stop the Spark context
sc.stop()
}
}
```

22. Consider a collection of 100 items of type integer given in the csv file. Write the Spark code to find the average of these 100 items.

CODE:

22

```
import org.apache.spark.{SparkConf, SparkContext}

object AverageCalculator {
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("AverageCalculator").setMaster("local")
val sc = new SparkContext(conf)

// Path to the input CSV file
val inputFile = "path/to/your/input.csv" // Replace with the actual path to your CSV file

// Step 2: Read the CSV file into an RDD
val lines = sc.textFile(inputFile)
```

```
// Step 3: Parse the RDD to extract integer values
val numbers = lines.map(line => line.toInt)

// Step 4: Use RDD operations to compute the sum and count of the integers
val sum = numbers.reduce(_ + _)
val count = numbers.count()

// Step 5: Calculate the average from the sum and count
val average = sum.toDouble / count

// Print the average
println(s"The average of the numbers is: $average")

// Stop the Spark context
sc.stop()
}
}
```

23. Consider a collection with items as (11,34,45,67,3,4,90).

i. Illustrate how spark context will construct the RDD from the collection, assuming number of partitions to be made is 3.

ii. Using mapPartitionsWithIndex return content of each partition along with partition index and apply a function, that increments the value of each element by 1, and returns an array.

CODE:

23

```
import org.apache.spark.{SparkConf, SparkContext}

object PartitionExample {
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("PartitionExample").setMaster("local")
val sc = new SparkContext(conf)

// Step 2: Create a collection
val data = Array(11, 34, 45, 67, 3, 4, 90)

// Step 3: Create an RDD from the collection with 3 partitions
val rdd = sc.parallelize(data, 3)

// Step 4: Use mapPartitionsWithIndex to return content of each partition along with partition index
// and increment each element by 1
val partitionedRDD = rdd.mapPartitionsWithIndex((index, iterator) => {
iterator.map(element => (index, element + 1))
})

// Step 5: Collect and print the results
val results = partitionedRDD.collect()
results.foreach { case (index, value) => println(s"Partition: $index, Value: $value") }

// Stop the Spark context
sc.stop()
}
}
```

24.

Consider the Item object.

Item = Map{(“Ball”:10), (“Ribbon”:50), (“Box”:20), (“Pen”:5), (“Book”:8), (“Dairy”:4),(“Pin”:20)}

Design the spark program to perform the following

i. Find how many partitions are created for the collection Item?

ii. Display the content of the RDD Display the content of each partition separately

CODE:

24

```
import org.apache.spark.{SparkConf, SparkContext}

object ItemRDD {
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("ItemRDD").setMaster("local")
val sc = new SparkContext(conf)

// Step 2: Create the Item map
val itemMap = Map("Ball" -> 10, "Ribbon" -> 50, "Box" -> 20, "Pen" -> 5, "Book" -> 8, "Dairy" -> 4,
"Pin" -> 20)

// Step 3: Create an RDD from the Item map
val rdd = sc.parallelize(itemMap.toSeq)
```

```
// Step 4: Find number of partitions created
val numPartitions = rdd.getNumPartitions
println(s"Number of partitions created: $numPartitions")

// Step 5: Display content of each partition separately
val partitionedRDD = rdd.mapPartitionsWithIndex { (index, iterator) =>
Iterator((index, iterator.toList))
}

// Collect and print the results
val results = partitionedRDD.collect()
results.foreach { case (index, partitionContents) =>
println(s"Partition $index:")
partitionContents.foreach(item => println(s" $item"))
}

// Step 6: Stop the Spark context
sc.stop()
}
}
```

25.

The table 1b provides the distributed data of the scala object

Item = Map{(“Ball”:10), (“Ribbon”:50), (“Box”:20), (“Pen”:5), (“Book”:8), (“Dairy”:4),(“Pin”:20)}

Partition1

{(“Ball”:10) (“Ribbon”:50) (“Box”:20)

Partition2

(“Pen”:5) (“Book”:8)

Partition3

(“Dairy”:4) (“Pin”:20)

Table 1b: Data distribution

Design the spark program to perform the following

i. Find how many partitions are created for the collection Item?

ii. Display the content of the RDD

iii. Display the content of each partition separately.

CODE:

```
import org.apache.spark.{SparkConf, SparkContext}

object ItemRDD {
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("ItemRDD").setMaster("local")
val sc = new SparkContext(conf)

// Step 2: Create the Item map
val itemMap = Map("Ball" -> 10, "Ribbon" -> 50, "Box" -> 20, "Pen" -> 5, "Book" -> 8, "Dairy" -> 4,
"Pin" -> 20)

// Step 3: Parallelize the Item map into an RDD with 3 partitions
val rdd = sc.parallelize(itemMap.toSeq, 3)

// Step 4: Find number of partitions created
val numPartitions = rdd.getNumPartitions
println(s"Number of partitions created: $numPartitions")

// Step 5: Display content of the RDD
println("Content of the RDD:")
rdd.collect().foreach(item => println(s" $item"))

// Step 6: Display content of each partition separately
println("\nContent of each partition separately:")
val partitionedRDD = rdd.mapPartitionsWithIndex { (index, iterator) =>
Iterator((index, iterator.toList))
}
partitionedRDD.collect().foreach { case (index, partitionContents) =>
```

```
println(s"Partition $index:")
partitionContents.foreach(item => println(s" $item"))
}

// Step 7: Stop the Spark context
sc.stop()
}
}
```

26.

Consider the text file words.txt as shown in the figure 1a.

Write the spark code to perform the following.

i) Count the number of occurrences of each word.

ii) Arrange the word count in ascending order based on Key.

iii) Display the words that begin with ‘s’.

She sells sea shells by the seashore

And the shells she sells by the seashore

Are sea shells for sure

Figure 1. a

```
import org.apache.spark.{SparkConf, SparkContext}

object WordCountExample {
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("WordCountExample").setMaster("local")
val sc = new SparkContext(conf)

// Step 2: Read the text file and split into words
val inputFile = "path/to/words.txt" // Replace with the actual path to your words.txt file
val lines = sc.textFile(inputFile)
val words = lines.flatMap(line => line.split("\\s+"))

// Step 3: Count the occurrences of each word
val wordCounts = words.map(word => (word.toLowerCase, 1))
.reduceByKey(_ + _)

// Step 4: Arrange word counts in ascending order based on key
val sortedWordCounts = wordCounts.sortByKey()

// Step 5: Display words that begin with 's'
val wordsStartingWithS = sortedWordCounts.filter { case (word, count) => word.startsWith("s") }
.collect()

// Step 6: Print the results
println("Word counts:")
wordCounts.collect().foreach { case (word, count) =>
println(s"$word: $count")
```

```
}

println("\nWord counts sorted in ascending order based on key:")
sortedWordCounts.collect().foreach { case (word, count) =>
println(s"$word: $count")
}

println("\nWords that begin with 's':")
wordsStartingWithS.foreach { case (word, count) =>
println(s"$word: $count")
}

// Step 7: Stop the Spark context
sc.stop()
}
}
```

27.

Illustrate the application of combineByKey to combine all the values of the same key in the following collection.

((“coffee”,2),(“cappuccino”,5),(“tea”,3),(“coffee”,10),(“ cappuccino”,15))

CODE:

27

```
import org.apache.spark.{SparkConf, SparkContext}
```

```
object CombineByKeyExample {
def main(args: Array[String]): Unit = {
// Step 1: Set up the Spark context
val conf = new SparkConf().setAppName("CombineByKeyExample").setMaster("local")
val sc = new SparkContext(conf)

// Step 2: Create the RDD with the given collection
val data = Seq(("coffee", 2), ("cappuccino", 5), ("tea", 3), ("coffee", 10), ("cappuccino", 15))
val rdd = sc.parallelize(data)

// Step 3: Apply combineByKey to combine values by key
val combinedRDD = rdd.combineByKey(
(value: Int) => value, // createCombiner: initialize the accumulator for each key
(acc: Int, value: Int) => acc + value, // mergeValue: add an element to the accumulator
(acc1: Int, acc2: Int) => acc1 + acc2 // mergeCombiners: combine accumulators from different partitions
)

// Step 4: Print the combined results
val results = combinedRDD.collect()
results.foreach { case (key, sum) =>
println(s"$key: $sum")
}

// Step 5: Stop the Spark context
sc.stop()
}
}
```