CSEE5590 Big Data Programming

In Class Programming –9 Report
(Jongkook Son)

Project Overview:

Apache Spark is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing.
Requirements/Task(s):

1. K-Means Clustering Algorithm
Review of the appraoch:
https://umkc.box.com/s/74rda02dy0uy1qpbasel5pl06gksf4wz
Review of Source Code:
https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/LocalKMeans.scala
2. Merge Sort Algorithm
Create a Map-Reduce Program to perform Merge-Sort Algorithm in Spark.
3. DepthFirst Search
Implement Depth First Search in Graph in Apache Spark

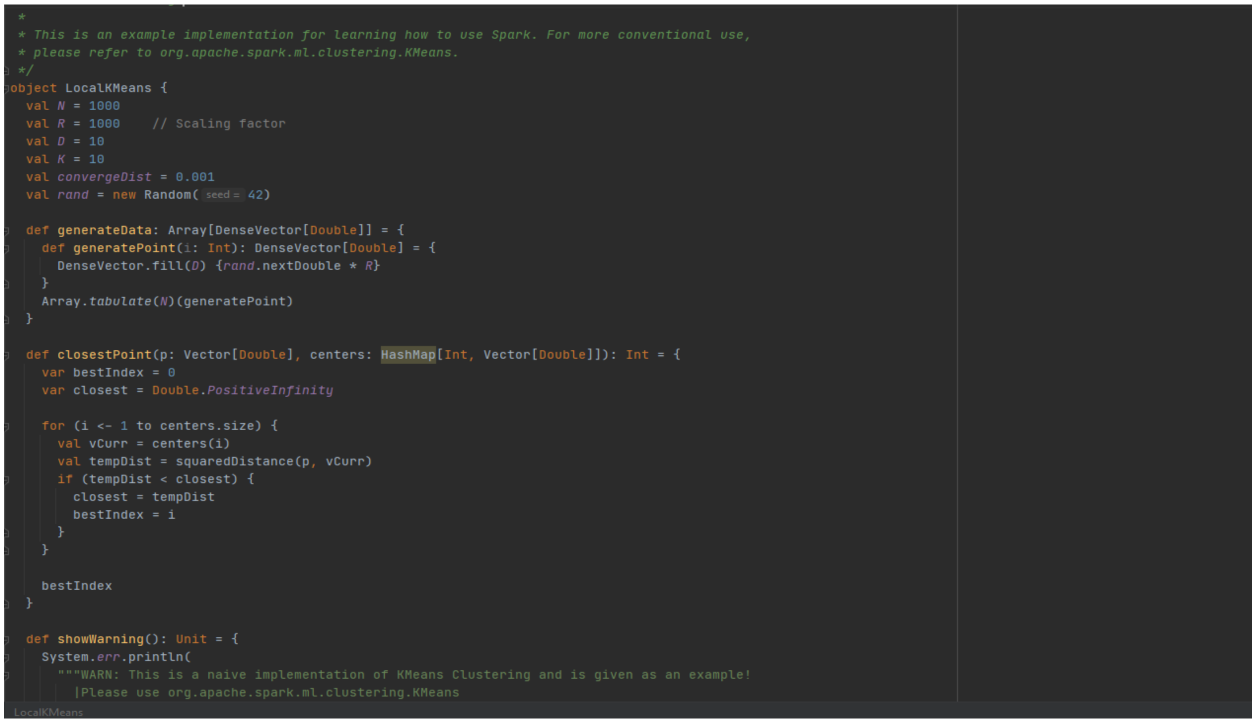
What I learned in ICP:

I could have learned how to set up Spark on my local machine and installing scala as a plugin for IntelliJ. In this ICP, We need to finish 3 tasks. Kmeans clustering, merge sort, depth first search all of these can be implemented by spark. Instead of spark mlib in the source code of the kmeans cluster they used breeze.linalg. Merge sort which is used by divide and conquer in which an input array is divided and an merge() function and dfs are implemented by spark . I could have practiced some use cases of spark.

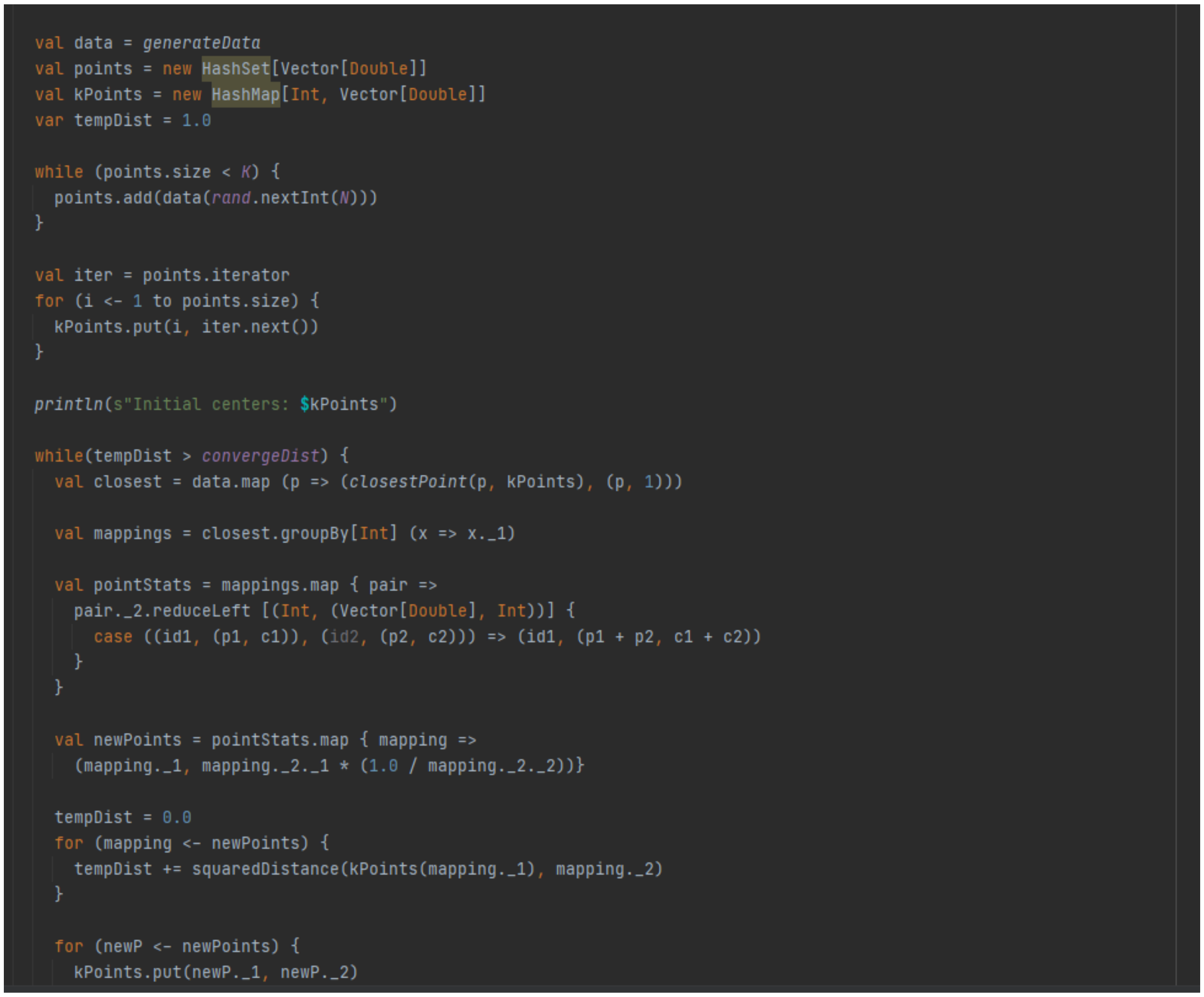
Task1> K-Means Clustering Algorithm

k-means is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters. The MLlib implementation includes a parallelized variant of the k-means++ method called kmeans.

Here I used source code that is given . Instead of using spark mlib In this source code We are going to use breeze.linalg.

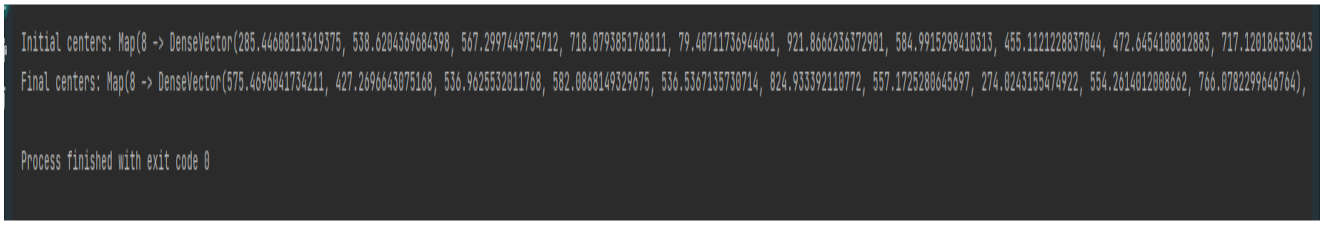


```
 *
 * This is an example implementation for learning how to use Spark. For more conventional use,
 * please refer to org.apache.spark.ml.clustering.KMeans.
 */
object LocalKMeans {
  val N = 1000
  val R = 1000    // Scaling factor
  val D = 10
  val K = 10
  val convergeDist = 0.001
  val rand = new Random( seed = 42)

  def generateData: Array[DenseVector[Double]] = {
    def generatePoint(i: Int): DenseVector[Double] = {
      DenseVector.fill(D) {rand.nextDouble * R}
    }
    Array.tabulate(N)(generatePoint)
  }

  def closestPoint(p: Vector[Double], centers: HashMap[Int, Vector[Double]]): Int = {
    var bestIndex = 0
    var closest = Double.PositiveInfinity

    for (i <- 1 to centers.size) {
      val vCurr = centers(i)
      val tempDist = squaredDistance(p, vCurr)
      if (tempDist < closest) {
        closest = tempDist
        bestIndex = i
      }
    }

    bestIndex
  }

  def showWarning(): Unit = {
    System.err.println(
      """WARN: This is a naive implementation of KMeans Clustering and is given as an example!
        |Please use org.apache.spark.ml.clustering.KMeans
```



```scala
val data = generateData
val points = new HashSet[Vector[Double]]
val kPoints = new HashMap[Int, Vector[Double]]
var tempDist = 1.0

while (points.size < K) {
  points.add(data(rand.nextInt(N)))
}

val iter = points.iterator
for (i <- 1 to points.size) {
  kPoints.put(i, iter.next())
}

println(s"Initial centers: $kPoints")

while(tempDist > convergeDist) {
  val closest = data.map (p => (closestPoint(p, kPoints), (p, 1)))

  val mappings = closest.groupBy[Int] (x => x._1)

  val pointStats = mappings.map { pair =>
    pair._2.reduceLeft [(Int, (Vector[Double], Int))] {
      case ((id1, (p1, c1)), (id2, (p2, c2))) => (id1, (p1 + p2, c1 + c2))
    }
  }

  val newPoints = pointStats.map { mapping =>
    (mapping._1, mapping._2._1 * (1.0 / mapping._2._2))}

  tempDist = 0.0
  for (mapping <- newPoints) {
    tempDist += squaredDistance(kPoints(mapping._1), mapping._2)
  }

  for (newP <- newPoints) {
    kPoints.put(newP._1, newP._2)
```

<Code>



```
Initial centers: Map(8 -> DenseVector(285.44608113619375, 538.6204369684398, 567.2997449754712, 718.0793851768111, 79.4071173694461, 921.8666236372901, 584.9915298410313, 455.1121228837044, 472.6454108812883, 717.12018653841
Final centers: Map(8 -> DenseVector(575.4696041734211, 427.2696643075168, 536.9625532011768, 582.0868149329675, 536.5367135730714, 824.933392110772, 557.1725280645697, 274.0243155474922, 554.2614012008662, 766.0782299646764),


Process finished with exit code 0
```

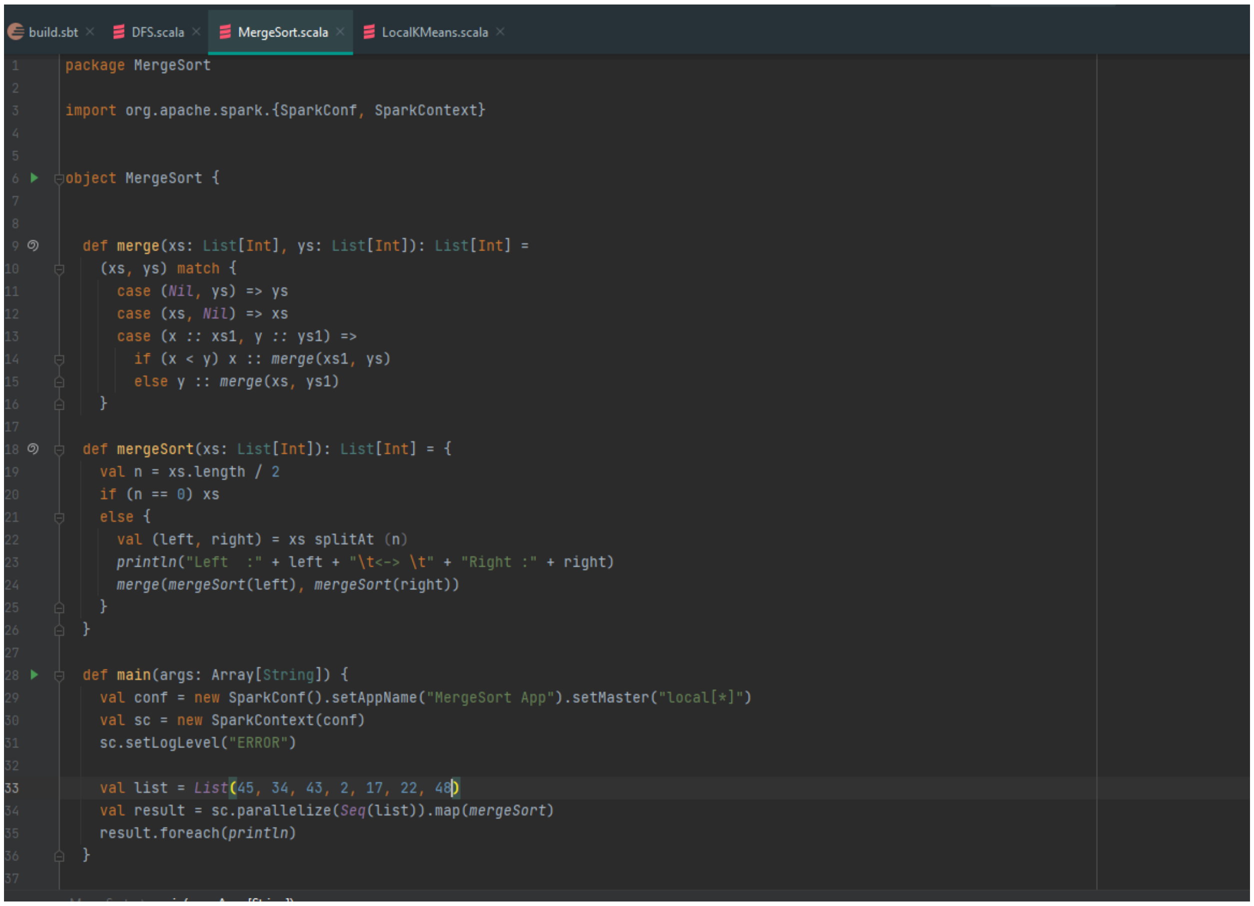
<Output>

Task2> Merge Sort Algorithm

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge () function is used for merging two halves. The merge (arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

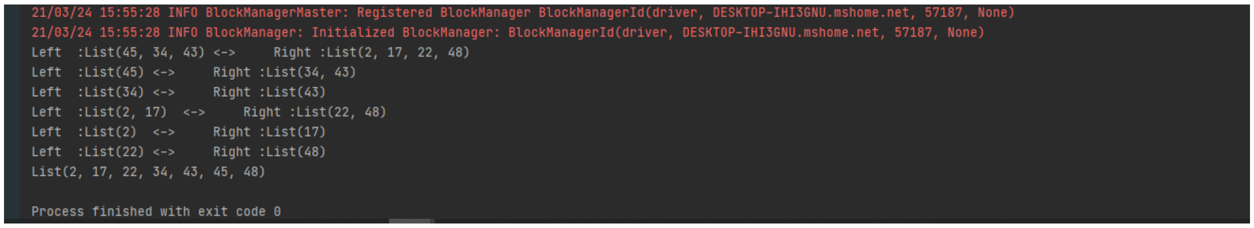
For this task, I had my data as a list of numbers. Then I did parallelization in order to use RDD.

here is the method for the merge sort algorithm



```scala
package MergeSort

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


object MergeSort {


  def merge(xs: List[Int], ys: List[Int]): List[Int] =
    (xs, ys) match {
      case (Nil, ys) => ys
      case (xs, Nil) => xs
      case (x :: xs1, y :: ys1) =>
        if (x < y) x :: merge(xs1, ys)
        else y :: merge(xs, ys1)
    }

  def mergeSort(xs: List[Int]): List[Int] = {
    val n = xs.length / 2
    if (n == 0) xs
    else {
      val (left, right) = xs splitAt (n)
      println("Left  :" + left + "\t<-> \t" + "Right :" + right)
      merge(mergeSort(left), mergeSort(right))
    }
  }

  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("MergeSort App").setMaster("local[*]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("ERROR")

    val list = List(45, 34, 43, 2, 17, 22, 48)
    val result = sc.parallelize(Seq(list)).map(mergeSort)
    result.foreach(println)
  }
```

<Code>



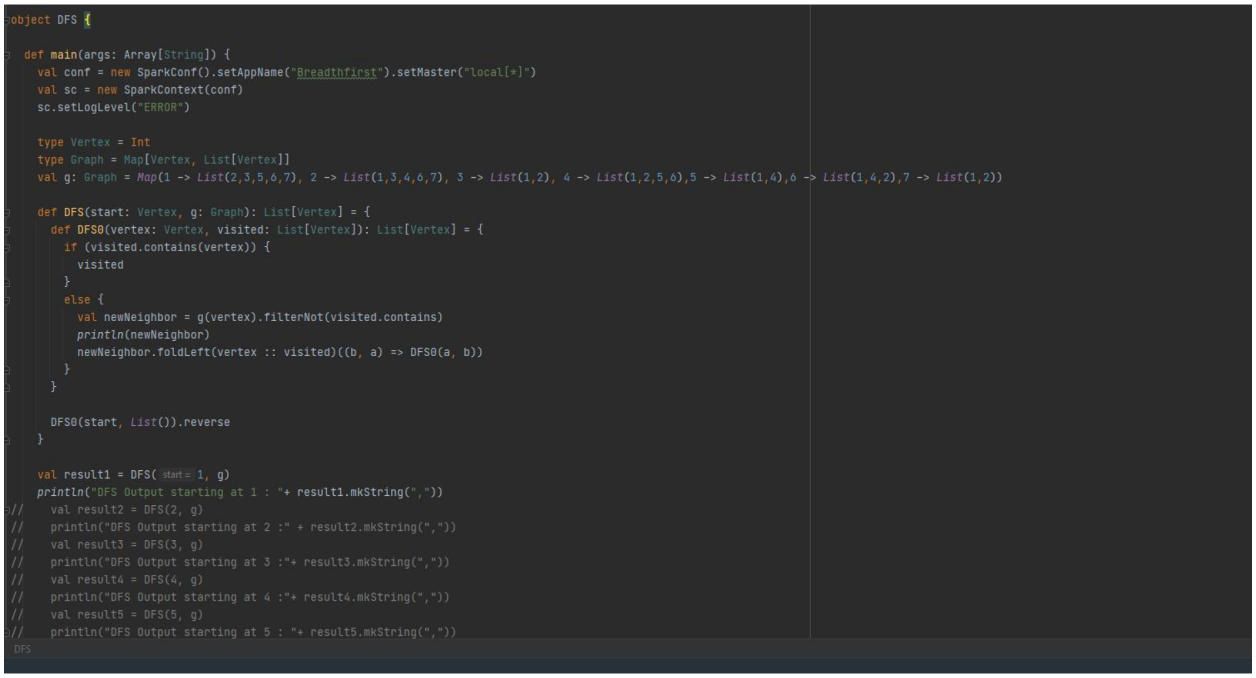
```
21/03/24 15:55:28 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, DESKTOP-IHI3GNU.mshome.net, 57187, None)
21/03/24 15:55:28 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, DESKTOP-IHI3GNU.mshome.net, 57187, None)
Left  :List(45, 34, 43) <->     Right :List(2, 17, 22, 48)
Left  :List(45) <->     Right :List(34, 43)
Left  :List(34) <->     Right :List(43)
Left  :List(2, 17)  <->     Right :List(22, 48)
Left  :List(2)  <->     Right :List(17)
Left  :List(22) <->     Right :List(48)
List(2, 17, 22, 34, 43, 45, 48)

Process finished with exit code 0
```

<Output>

Task3> DFS Algortihm

The depth-first search algorithm allows us to determine whether two nodes, node x and node y, have a path between them. The DFS algorithm does this by looking at all of the children of the starting node, node x, until it reaches node y. The depth-first algorithm sticks with one path, following that path down a graph structure until it ends.

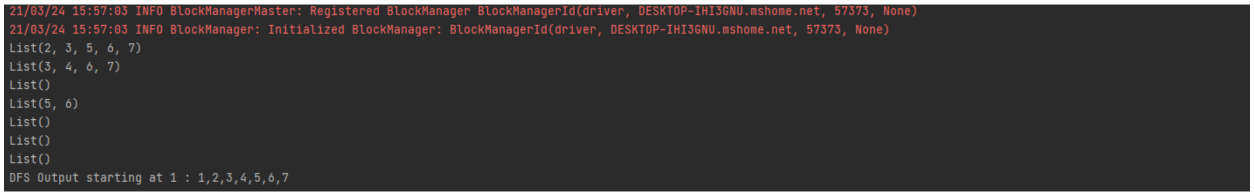
For this task, I also used a method to get the vertexes and their lists in a graph. Then it will get the visited points by declaring the start point.



```scala
object DFS {

  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Breadthfirst").setMaster("local[*]")
    val sc = new SparkContext(conf)
    sc.setLogLevel("ERROR")

    type Vertex = Int
    type Graph = Map[Vertex, List[Vertex]]
    val g: Graph = Map(1 -> List(2,3,5,6,7), 2 -> List(1,3,4,6,7), 3 -> List(1,2), 4 -> List(1,2,5,6),5 -> List(1,4),6 -> List(1,4,2),7 -> List(1,2))

    def DFS(start: Vertex, g: Graph): List[Vertex] = {
      def DFS0(vertex: Vertex, visited: List[Vertex]): List[Vertex] = {
        if (visited.contains(vertex)) {
          visited
        }
        else {
          val newNeighbor = g(vertex).filterNot(visited.contains)
          println(newNeighbor)
          newNeighbor.foldLeft(vertex :: visited)((b, a) => DFS0(a, b))
        }
      }

      DFS0(start, List()).reverse
    }

    val result1 = DFS( start = 1, g)
    println("DFS Output starting at 1 : "+ result1.mkString(","))
//    val result2 = DFS(2, g)
//    println("DFS Output starting at 2 :" + result2.mkString(","))
//    val result3 = DFS(3, g)
//    println("DFS Output starting at 3 :"+ result3.mkString(","))
//    val result4 = DFS(4, g)
//    println("DFS Output starting at 4 :"+ result4.mkString(","))
//    val result5 = DFS(5, g)
//    println("DFS Output starting at 5 : "+ result5.mkString(","))
```

DFS

<Code>



```
21/03/24 15:57:03 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, DESKTOP-IHI3GNU.mshome.net, 57373, None)
21/03/24 15:57:03 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, DESKTOP-IHI3GNU.mshome.net, 57373, None)
List(2, 3, 5, 6, 7)
List(3, 4, 6, 7)
List()
List(5, 6)
List()
List()
List()
DFS Output starting at 1 : 1,2,3,4,5,6,7
```

<Output>