Problem Set 1

# **Exercise 1**

## a).

. We know that , therefore the probability of no machines fails during the time is:

## b).

## c).

We know that , therefore , we can also demonstrate the result by calculating the sum directly. We know

Then

# **Exercise 2:**

## a)

The descriptions usually come from the spark official website: [spark.apache.org](https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html). The description of the transformations of the join(), sort() (sortby or sortByKey) and groupby() are as follows:

1. join()

join(otherDataset, [numTasks]): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Input is pairs of the dataset, output is also pairs of the dataset with the data of the same key value joined together. See example:

Input:



Output:



1. sort()

We did not find sort() on the spark.apache website, assuming here it means sortby() or sortByKey(). The description of sortBy() and sortByKey() is as follows:

* sortByKey(self, ascending=True, numPartitions=None, keyfunc=lambda x: x): Sorts this RDD, which is assumed to consist of (key, value) pairs.

Input:



Output:



* sortBy(self, keyfunc, ascending=True, numPartitions=None): Sorts this RDD by the given keyfunc, default is an ascending order

Input:



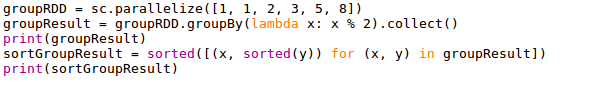
Output:



1. groupBy()

groupBy(self, f, numPartitions=None)，Return an RDD of grouped items， here f is a function handler.

Input:

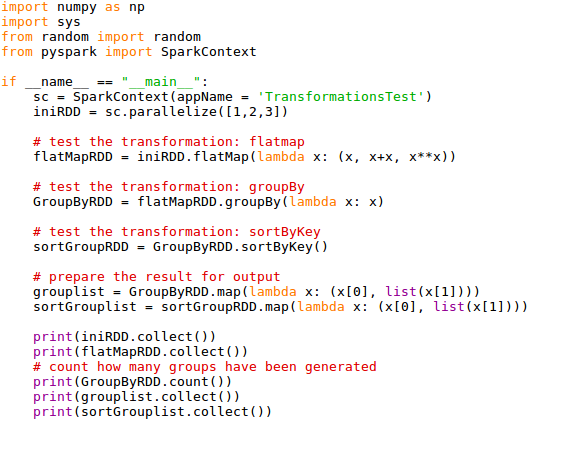


Output:

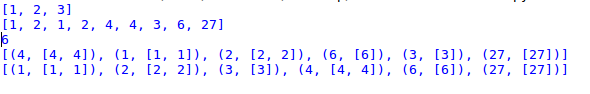


## b)

Three transformation examples are given: flatmap, groupby, sortByKey

The source code is as follows: 

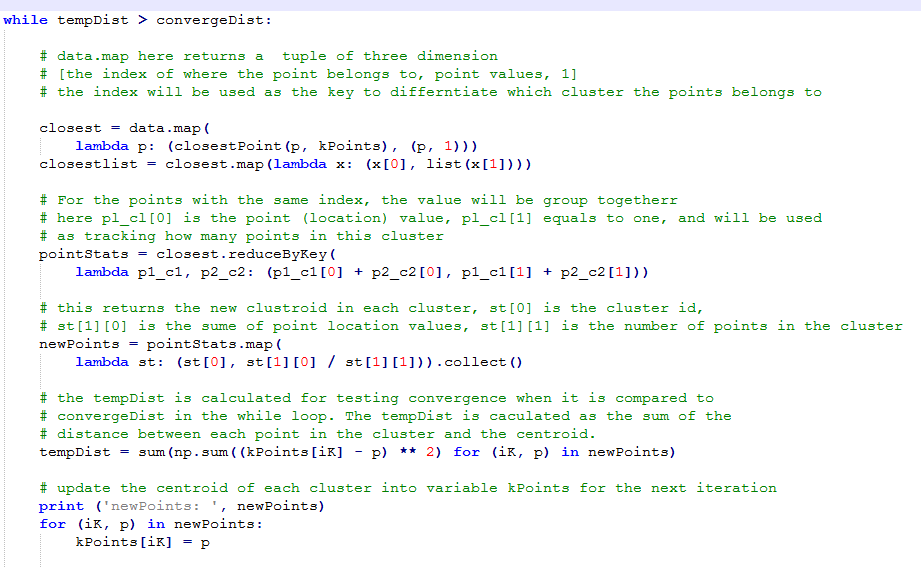
The result is as follows:



# **Exercise 3:**

## a)

The description of each step in the for-loop is given out in the comment part of the code. The detailed explanation could be referred in the source code.



### b)

Please refer to the source code to see how GroupByKey is implemented. The scalability of the solution by GroupByKey is bad. That is because when calling GroupByKey, all the key-value pairs are shuffled around, and a lot of unnecessary data is being transferred over the network. When the data is large, and using GroupByKey may load the whole array of values with the same key into memory and it may cause a lot of memory issue.

(for reference: )

<https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html>

## **Exercise 4:**

a).

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

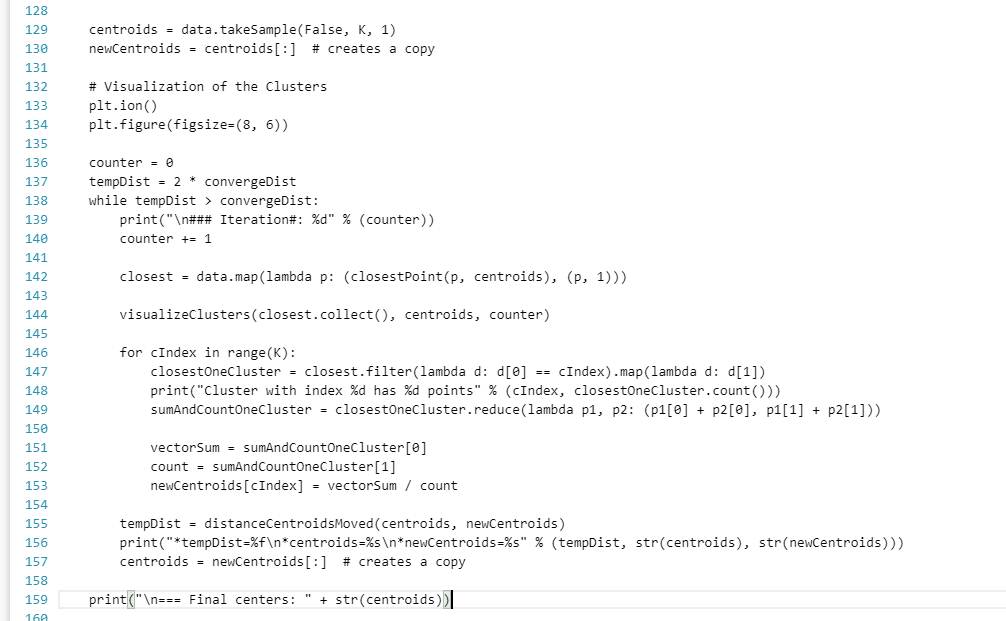
After the broadcast variable is created, it should be used instead of the value v in any functions run on the cluster so that v is not shipped to the nodes more than once. In addition, the object v should not be modified after it is broadcast in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

(refer to <http://spark.apache.org> )

b).

## **Exercise 5:**

The code is as follows:



There are three cases:

Case 1: on line 130: newCentroids = centroids[:] , and line 157: centroids = newCentroids[:]

Case2: on line 130: newCentroids = centroids[:], and line 157: centroids = newCentroids

Case3: on line 130: newCentroids = centroids, and line 157: centroids = newCentroids[:]

Case4: on line 130: newCentroids = centroids, and line 157: centroids = newCentroids

In Case 1, the original case: in line 130, the initial centroids value is copied to newCentroids, in the for loop, newCentroids is updated and therefore the following tempDist is updated, and in line 157, the updated newCentroids value is copied back to centroids.

In Case 2, in line 130, the initial centroids value is copied to newCentroids. In the first and second for loop, newCentroids is updated. However, in line 157, the reference of newCentroids is copied to centroids, which will refer to the same list as newCentroids refer to. This means in the second iteration, in line 155 when calculating the tempDist, both newCentroids and centroids refer to the same list, and tempDist will be zero and the while loop will be stopped. Thus, the result will be different, the iteration will be implemented only twice. Also the result of centroids is also only updated twice.

In Case 3, in line 130 the list which centroid refers to is copied so that newCentroids will refer to the same list. In the first iteration, newCentroids is updated, which means the value in the list which centroid refers to is also updated. The tempDist will have value of zero when it is calculated for the first time. In line 157, the list value which newCentroids is refered to will be copied to centroids, now newCentroids and centroids refers to different list and have the same value. But the while loop is stopped because of the tempDist is equal to zero after the first iteration. Case 1, Case 2 and Case 3 will have different result. In Case 3, the iteration is only implemented once, and also the result of centroids is updated once.

In Case 4, the result will be same as Case three until line 157. While in 157, the reference of newCentroids is copied back to centroids, which means they refer to the same list. As to the result, Case 4 will have the same result as Case 3, i.e. the iteration is implemented only once and centroids is updated once.