Project 8 (Java): K-means clustering. Given a list of 2-D points, and a cluster number, K, the task is to partition the input point set to K clusters such that all points within one cluster are closer to the centroid of their own cluster (in distance) than to the centroids of all other clusters.

Algorithm steps for this project:

```
Step 0: pointSet ← given, K ← given
Step 1: randomly selects k points from pointSet. (The selected k points will be the initial K centroids.)
Step 2: change ← 0
Step 3: For every point, p, in pointSet, computes distance from p to each centroid (K of them).
        So, we have K distances: d1, d2,..., dk, associate with each di is the label of the centroid.
Step 4: minLabel ← the label of the minimum distance among d1, d2,..., dk
Step 5: if p's label != minLabel
                P's label ← minLabel
                changes ++
Step 6: repeat step 3 to step 5 until all points in pointSet are processed
Step 7: if changes > 2
            Compute K centroids
Step 8: repeat step 2 to step 7 while change > 2
```

*Remark: YOU MUST IMPLEMENT THE ALGORITHMS AS GIVEN In the above;
otherwise, you will receive 0 pt for this project, if you USE different ALGORITHM!

*** What you need to do for this project:

1. You will be given a data file to test your program.
2. Run your program four times using K = 3, 4, 5
3. Include in your pdf hard copies (in pdf file):
   - Cover page
   - Source code
   - Print outFile1 and outFile2 for K = 3 (with proper caption)
   - Print outFile1 and outFile2 for K = 4 (with proper caption)
   - Print outFile1 and outFile2 for K = 5 (with proper caption)

****
Language: Java
****
Project points: 10pts

Due Date: Soft copy (*.zip) and hard copies (*.pdf):

+1 (11/10 pts): early submission, 5/6/2022 Friday before midnight.
-0 (10/10 pts): on time, 5/10/2022 Tuesday before midnight.
-1 (9/10 pts): 1 day late, 5/11/2022 Wednesday before midnight.
-2 (8/10 pts): 2 days late, 5/12/2022 Thursday before midnight.
(-10/10 pts): none submission, 5/12/2022 Thursday after midnight

*** Name your soft copy and hard copy files using the naming convention as given in the project submission requirement.

*** All on-line submission MUST include Soft copy (*.zip) and hard copy (*.pdf) in **the same email attachments** with correct email subject as stated in the email requirement; otherwise, your submission will be rejected.

****
I. input:
****
- inFile (args [0]) : a text file with the following format:
  The first text line is the dimension (#of rows and # of columns) of the image,
  The second text line is number of points in the image, then
  follows by a list of points, in x-y coordinates.

For example:

```
40  50  // The image has 40 rows and 50 columns
25        // There are 25 object points in the image
12  30 // A point on row 12 and column 30
10  21 // A point on row 10 and column 21
:
:
```

- K (args [1]) // for the clustering in K groups

****
II. Outputs:
****
- outFile1 (args[2]): 2D displays of the result of clustering after each iteration.

*** use "`courier new`" font, so the output of your 2D arrays all line-up row by row and column by column.

For example (the clustering results of the input in the above):

```
*** Result of iteration 1 **** // Use courier new font

   1 1 1
 1 1 1 2
   2 2
         1 2
       1 2 2
       1 2 2
       2 2 2
:
:
:
:
*** Result of iteration 4 ****

   1 1 1
 1 1 1 1
   1 1
         2 2
       2 2 2
       2 2 2
       2 2 2
```

- outFile2 (use args [3]): The clustering result -- a txt file representing the list of 2D points and their labels in the following format:
  The first text line is the dimension of the image
  The second text line is the number of points in the list
  The third text line and after are 2D points' x-y coords and points' labels.

For example:

```
40  50          // The image has 40 rows and 50 columns
25              // There are 25 data points in the point set
12  30  3       // A point on row 12 and column 30 with label 3
10  21  2       // A point on row 10 and column 21 with label 2
:               // etc.
:
:
:
```

****
III. Data structure:
****

- A Kmean class
    - A Point class:
        - (double) Xcoord // convert to (int) when plotting onto 2D displayAry
        - (double) Ycoord // convert to (int) when plotting onto 2D displayAry
        - (int) Label // initialize to 0
        - (double) Distance // the distance to its own cluster centroid; initialize to 99999.00

    - (int) K // K clusters; given in args
    - (int) numPts (int) // The total number of points
    - (Point) pointSet [numPts] // 1D array of Point class of size numPt; to be dynamically allocated during run-time
        // initialized to 99999.0 a large distance, for all points
    - (int) numRows
    - (int) numCols
    - (** int) displayAry // a 2D array, size of numRows by numCols for displaying purposes
    - (Point) KcentroidAry [K+1] // 1D array of Point class of size K to store the info of centroids;
        // to be dynamically allocated in class constructor, size of K+1
        // we do NOT want to use 0 as cluster label so the cluster label
        // will run from 1 to K, therefore, the size of array is K+1
        // Distance is set to 0.0 distance from itself.
    - (int) change // for tracking the label changes, initialize to 0

    Methods:
    - constructor (…) // does all dynamically allocations, initializations, can loadPointSet () also.
    - loadPointSet (...) // load data points in pointSet, on your own. Remember, each data point has 4 attributes!
    - kMeansClustering (...) // see algorithm below
        // Implementation steps follow the algorithm steps given in the above.
    - selectKcentroids (...)// see algorithm below
    - computeCentroids (...) // see algorithm below
    - (int) DistanceMinLabel (...) // see algorithm below
    - (double) computeDist (...) // On your own
        // Compute the distance from pt to a given centroid.
        // The method returns the distance
        // YOUR SHOULD KNOW HOW TO WRITE THIS METHOD!
    - PlotDisplayAry (pointSet, displayAry) // On your own
        // First, you need to reset displayAry to zero, then for each point i, in pointSet, plot the pointSet[i].Label
        // onto the displayAry at the location of pointSet[i]'s Xcoord and Ycoord; need to convert pointSet[i]'s
        //Xcoord and Ycoord to integer to get the 2D coordinates
        // And, also display the K-controids, use 'A' for centroid 1, 'B' for centroid 2, 'C' for 3, etc.
    - PrettyPrint (displayAry, outFile1, iteration) // On your own
        // write Caption indicating which K is used and which iteration
        // then, write the displayAry to outFile1 as follows:
        // if displayAry (i,j) > 0
            print displayAry (i,j)
            else print one blank space.

    - PrintResult (pointSet, outFile2) // On your own
    // Output numRows numCols, numPts, and the x-coord, y-coord in
    // pointSet, and label, using the outFile2 file format given in the above

```
*******************************
IV. main (...)  // The algorithm may contain bugs, debugging is yours
*******************************
Step 1:
        inFile ← Open from args[0]
        K ← from args [1]
        outFile1, outFile2 ← Open from args[]
        numRows, numCols ← read from inFile.
        numPts ← read from inFile
        displayAry ← Dynamically allocate a 2-D arrays, size numRows by numCols.
        pointSet ← Dynamically allocate the pointSet array, size of numPts
        KcentroidAry ← Dynamically allocate centroids struct, size of K+1
        loadPointSet (inFile, pointSet)
Step 2: kMeansClustering (pointSet, K, KcentroidAry)
Step 3: close all files
*******************************
VI. kMeansClustering (pointSet, K, KcentroidAry) // This algorithm may contain bugs, debugging is yours
*******************************
Step 0: iteration ← 0
Step 1: selectKcentroids (pointSet, K, KcentroidAry) // initial selection of K centroids, ramdonly.
Step 2: index ← 0
            iteration++
Step 3: PlotDisplayAry (pointSet, displayAry)
            PrettyPrint (displayAry, outFile1, iteration)
Step 4: change ← 0
Step 5: pt ← pointSet [index]  // get next pt, which is a Point class data type!!
            minDist ← pointSet[index]'s distance
Step 6: minLabel ← DistanceMinLabel (pt, KcentroidAry, minDist)
Step 7: if pointSet[index]'s label != minLabel
                        pointSet[index]'s label ← minLabel
                        pointSet[index]'s distance ← minDist
                        changes ++
Step 8: index ++
Step 9: repeat step 5 to step 8 while index < numPts
Step 10: if changes > 2
                computeCentroids (pointSet, KcentroidAry)
Step 11: repeat step 2 to step 9 until change <= 2

*******************************
VII. selectKcentroids (pointSet, K, KcentroidAry)
*******************************
// To insure not to select two identical centroids,
// you may use an array or any data structure to store
// the indexes that are generated prior, so to check the current
// randomly generated index to see whether it has been selected prior;
// you may also write a local checkRepeat(...) method;
// checkRepeat returns true is the index has been generated prior
// and returns false otherwise

Step 0: Kcnt ← 0
Step 1: index ← randomly select an index from 0 to numPts-1
                // Call a random generator function for this
```

```
Step 2: repeatYN ← checkRepeat (index,...)
Step 3: repeat step 1 to step 2 until repeatYN is false
Step 4:  KCnt ++
                    KcentroidAry [Kcnt].Xcoord ←  pointSet[index].Xcoord
                    KcentroidAry [Kcnt].Ycoord ←  pointSet[index].Ycoord
                    KcentroidAry [Kcnt].label← Kcnt          // its own label
                    KcentroidAry [Kcnt].Distance ← 0.0

Step 5: repeat step 1 to step 4 while Kcnt < K
```

****
VIII. (int) DistanceMinLable (pt, KcentroidAry, minDist)
****

```
// compute the distance from a point pt to each of the K centroids. The method returns minLable
// This algorithm may contain bugs, debugging is yours

Step 0: minDist ← 99999.00
           minLabel ← 0
Step 1: label ← 1
Step 2: whichCentroid ← KcentroidAry[label] //whichCentroid is a Point class type
Step 3: dist ← computeDist (pt, whichCentroid)// pt is a Point class type.
Step 4: if dist < minDist
                         minLabel ← label
                         minDist ←dist
Step 5: label ++
Step 6: repeat step 2 to step 5 while label <= K
Step 7: return minLabel
```

****
IX. computeCentroids (pointSet, KcentroidAry)
****

```
// Go thru the entire pointSet array only once to compute the centroids for all
//K clusters. Store the computed centroids in each KcentroidAry [i], i = 1 to K.

Step 0: (double) sumX[] ← dynamically allocate 1-D array, size of K+1
                              // initialize to 0.0
            (double) sumY[] ← dynamically allocate 1-D array, size of K+1,
                              // initialize to 0.0
            (int) totalPt[] ← dynamically allocate 1-D array, size of K+1
                    // initialize to 0
Step 1: index ← 0
Step 2: label ← pointSet[index].label // get the point's cluster label
              sumX[label] += pointSet[index].Xcoord
              sumY[label] += pointSet[index].Ycoord
              totalPt[label] ++
Step 3: index++
Step 4: repeat step 2 to step 3 while index < numPts
step 5: label ← 1
step 6: if totalPt[label] > 0.0
                    KcentroidAry [label].Xcoord ←(sumX[label]/ totalPt[label])
                    KcentroidAry [label].Ycoord ←(sumY[label]/ totalPt[label])
Step 7: label ++
Step 8: repeat step 6 to step 7 while label < K // maybe <=
```