Question	Status	Comment
Q1	Fully Working	
Q2	Fully Working	Reoptimized it to run quicker

Datasets

To make the datasets simply run:

`Datasets.java`

No parameters are needed and it will create the `points` and `rectangles` files which are both at least 100 MB. The points dataset has random numbers from 1 to 10000 for both x and y. The rectangles dataset is slightly different. The x and y are from 1 to 9999 because it is impossible to have a rectangle with a point starting at 10000. Also the height and width are randomized from 1 - 20 because having large rectangles will greatly impact the performance of the spatial join job. The logic will still work with larger rectangles its just for execution time that we constrained the height and width. Also asked the professor to ensure I can do this.

Question 1

How to run the Spatial Join

`SpatialJoin.java pointsFile rectangleFile outputFile x1 y1 x2 y2`

Here the x and y's are the spatial window for the program. You can run the program without these parameters and it will use the default `1 1 10000 10000`.

How does this job work. The main idea is to divide the entire space into equal subsections. In this case will be 10 x 10. Inside each of the subsections is where we determine points that are in each rectangle. The reasoning for dividing into these subsections is that we don't need to see other subsections to determine if a point belongs to a rectangle.

Point Mapper:
If the point is not inside the spatial window then we ignore it and move on. If it is then we determine the section is belongs to. We do this by rounding down both the x and y to the nearest 10. For example `(16,402) -> (10,400).` We set the key to be the x and y of this subsection and the value to be the original x and y and what dataset it is. So in this case: `<(10, 400), (P, 16, 402)>`

Rectangle Mapper

We first check if any part of the rectangle is within the spatial window. If so then we can continue. A rectangle can be a part of multiple sections so we need to find all of them. We do this by doing the same round down to the nearest 10 for the bottomLeftX and bottomLeftY and finding the subsection for this. Then we increment both the x and y by the size and do the same rounding down to find all of the sub-sections. Each time we add the key-value pair <subsection, rectangle>

Reducer

Each reducer deals with one of the subsections which will contain all of the rectangles and points in the section. We store all of the points and rectangles into a list. Then for each rectangle, we can simply check if a point is inside the rectangle and if it is then write to the output.

Example Output
Spatial window: 1 5 10 20

```
(2,1,14,6),(7,7)
(2,1,14,6),(3,5)
(4,2,6,6),(7,7)
(4,2,6,6),(8,7)
(4,2,6,6),(9,6)
(3,3,17,17),(7,7)
(3,3,17,17),(8,7)
(3,3,17,17),(9,6)
(7,6,18,12),(8,7)
(3,3,17,17),(7,12)
(3,3,17,17),(7,12)
(3,3,17,17),(7,19)
(3,3,17,17),(8,19)
(3,3,17,17),(5,15)
(3,3,17,17),(9,17)
(3,3,17,17),(9,18)
(7,6,18,12),(8,19)
(7,6,18,12),(9,17)
(7,6,18,12),(9,18)
(1,17,16,11),(2,18)
(1,17,16,11),(7,19)
(1,17,16,11),(8,19)
(1,17,16,11),(9,18)
(1,17,16,11),(2,18)
(2,1,14,6),(7,12)
(2,1,14,6),(7,12)
(2,1,14,6),(3,14)
(6,18,9,2),(7,19)
(1,17,16,11),(6,20)
```

Question 2

How to run Outlier Detection:

`OutlierDetection.java pointsFile outputFile r k`

This job works similar to Question 1. We will be making subsections of the space. Within each subsection we take a point and count up all of the neighbors it has and if it has less than k neighbors within r then it is considered an outlier. However we have to modify the subsections so that we can account for all of the neighbors of distance r that wouldn't normally be in the same subsection. The idea is that subsections can overlap with each other so that a point can go to multiple sections. We can think that instead of it being a point it is more like a circle with radius r. We check if the circle belongs to a section and if it does then put it in there. This way we can guarantee that for a point that is within the section all of its neighbors in range r would also exist in the same location.

Point Mapper:

We need to decide what subsections a point belongs to. So we create a circle from the point and get all of the chunks that are related to that point excluding those that are out of the space. This means that one point can go to multiple chunks. The key-value pair is <chunk, point>.

Reducer:

We first need to create a list of all the points. Then we have to calculate the neighbors for each point. We first need to make sure that the point is part of the main subsection, because we only have all of the neighbors for the points inside the main chunk. Then we count the number of neighbors a point has by calculating the euclidean distance between the two points and if it is less then the radius then we add it. We finally only write to the output if the total number of neighbors is less than k.

Output with r = 10 and k = 10

```
2,6667
2,6667
1,706
1,706
1,7393
1,7377
1,7442
1,7456
2,7444
1,7443
1,7997
2,9024
1,9573
2,9780
1,9769
3,9998
28,9999
1,9912
1,9914
202,1
2248,1
2251,2
2265,1
3168,1
3250,9998
3255,9998
3855,9999
4448,1
4452,1
4606,202
4608,202
4857,1
4855,1
512,7002
5124,9998
5299,1
5301,9999
5797,2
6344,1
7088,1
7190,9998
7864,2
7815,245
8269,1
8507,1
8552,1
8556,2
8644,2
8641,2
9408,1
9801,2
9999,1509
9998,2574
9999,3718
9999,9980
9999,9979
```

Something to note is that most of the points are along the edge of the world. This makes sense because there can’t be any points there so those points have less of a chance to have many neighbors.