Report

Calibre D2S R&D C++ Project 2021

By:

Aya Tarek El-Ashry

August 2021

Table of Contents

Platform Used

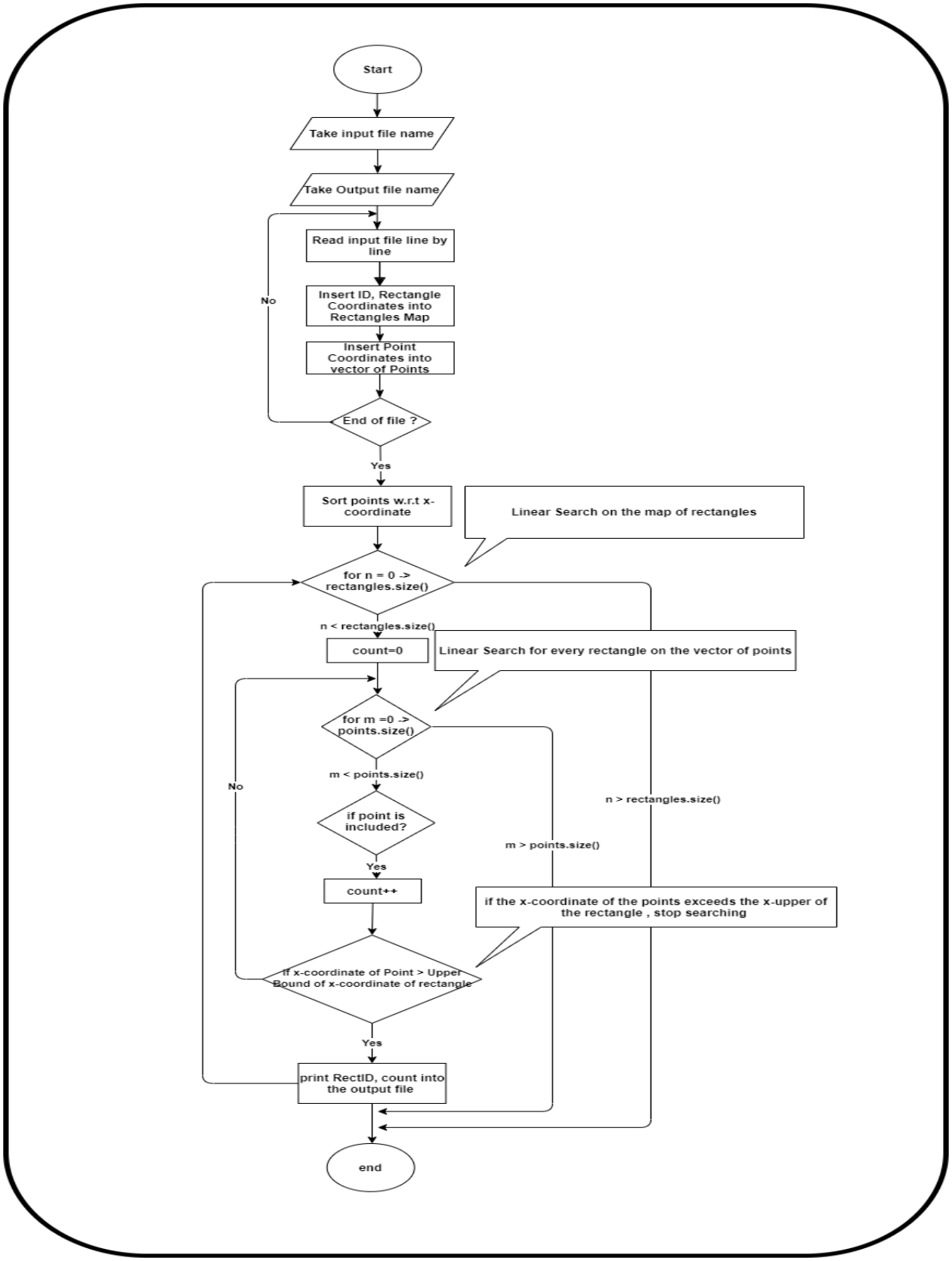
I used Visual studio 2017 running on windows 10.

To run on windows, copy windows-version source files into visual studio 2017 new project.

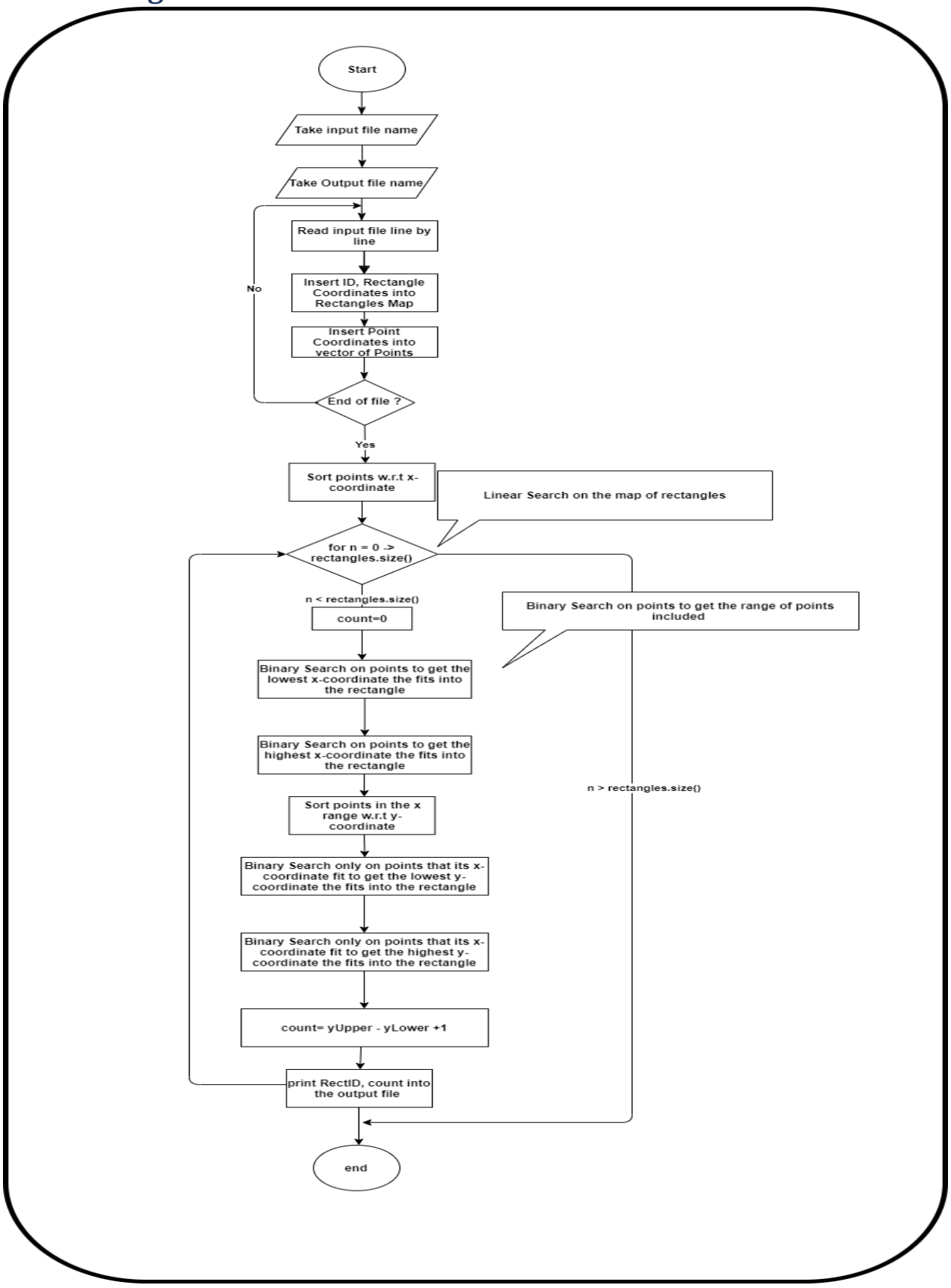
And while running, input the name of the input file that located to the same directory & the name of the output file that you want to be created (it will be created in the same directory).

Flow Chart Explaining the algorithm

- **1st algorithm**



➢ 2nd algorithm



Complexity in Big O notation

➔Both algorithms have the same complexity of reading data & inserting it into map of rectangles & vector of points.

But for the algorithm of Searching and Calculating the number of points included itself:

- **1st algorithm:**

Complexity = O (n*m)

Where n is the total number of rectangles

& m is the total number of points

- **2nd algorithm:**

Complexity = O (n * log(m) * log(k))

Where n is the total number of rectangles

& m is the total number of points

& k is the maximum number of points included within the x-coordinate range in a rectangle

Algorithm Explanation

- ## Pseudocode

➢ 1st algorithm:

Read file and put all input rectangles in a map of (RectID, Rectangle) & al the input points into a vector of Points.

Sort vector of points according to x-coordinate.

```
For i =0 -> rectangles_Map.size()
do
        Count=0
        For j =0 -> Points.size()
        do
                If (point is included with the rectangle or on its edges)
                Count++
                If (x-coordinate of the point > Upper x-coordinate of the
        rectangle)
                        Break
        end
        Print rectID and the number of points included within it in the
        output file.

end
```

➢ 2nd algorithm:

Read file and put all input rectangles in a map of (RectID, Rectangle) & al the input points into a vector of Points.
Sort vector of points according to x-coordinate.
For i =0 -> rectangles_Map.size()
do
count=0
//to get the lowest x-coordinate the fits into the rectangle
Binary Search to get the lower bound of x-coordinate that fits into the rectangle.
//to get the highest x-coordinate the fits into the rectangle
Binary Search to get the upper bound of x-coordinate that fits into the rectangle.
////////////////////// No range of x-coordinates fit in the rectangle //////////////////////
If (xLower == xUpper == Points.size())
Count = 0
Else
Create vector of points that only includes the range of points fit in the x-axis.
Sort this new vector of points according to y-coordinate.
//to get the lowest y-coordinate the fits into the rectangle
Binary Search to get the lower bound of y-coordinate that fits into the rectangle.
//to get the highest y-coordinate the fits into the rectangle
Binary Search to get the upper bound of y-coordinate that fits into the rectangle.

////////////////// no range of y-coordinates fit in the rectangle //////////////////
If (ylower_bound >= yUpper_bound)
Count=0
Else
Count = (yUpper_bound - ylower_bound) +1

Print rectID and the number of points included within it in the output file.
End

- **Visual drawing**

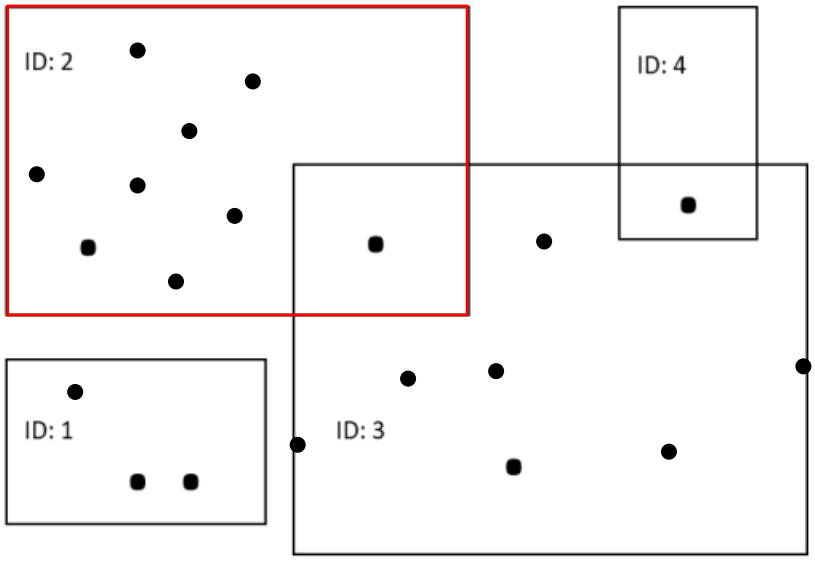
➢ **1st algorithm**

1. **Looping on every rectangle**

1.1 for rectID= 1 -> number of rectangles

for rectID =2

➔first initialize the count to zero

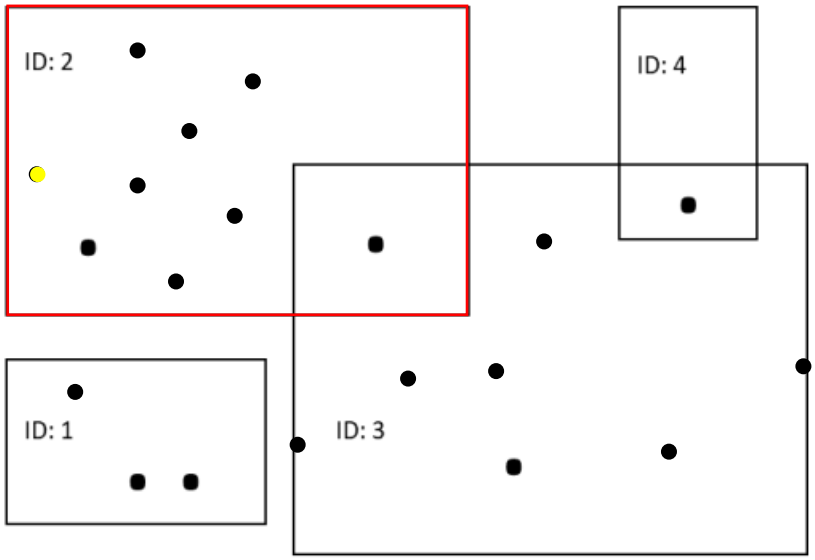


1.1.1 Sort the points with respect to x-coordinate.

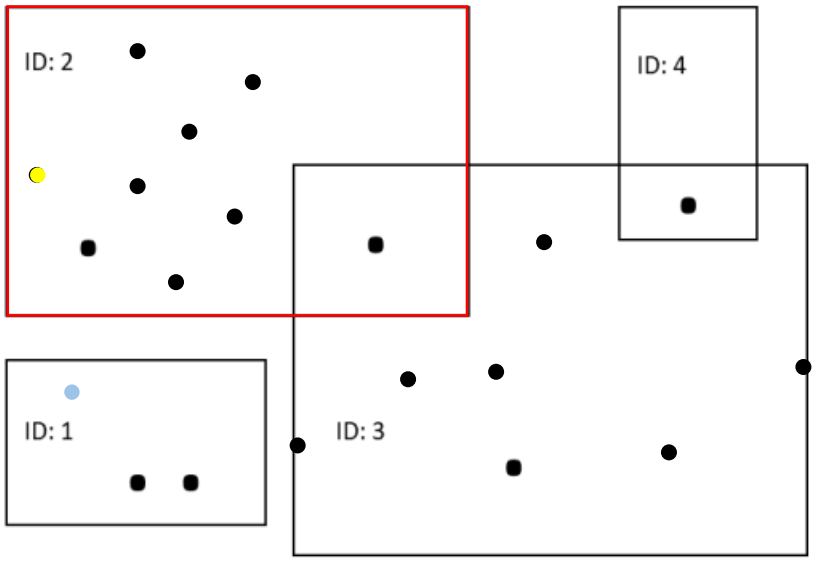
1.1.2 Loop through the vector of points.

& If included ➔ count++

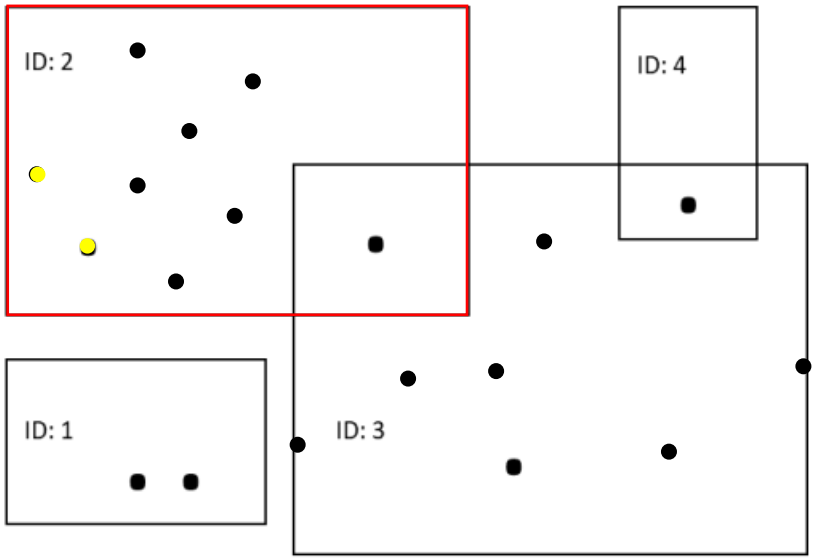
, Otherwise Skip.



Count++

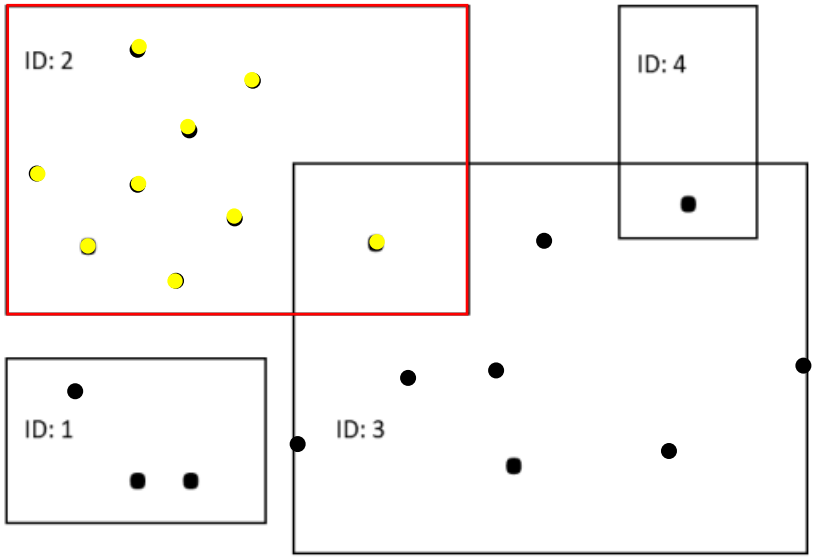


Skip



Count++
.. etc

1.1.2.1 if you find a point with x > upper x in the rectangle.
break



Print the rectID, count of points included in the output file.

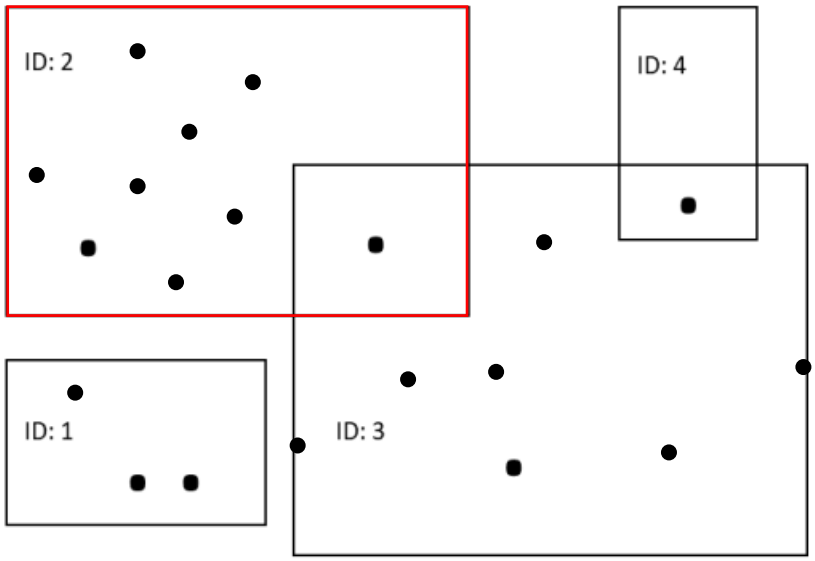
➢ 2nd algorithm

2. Looping on every rectangle

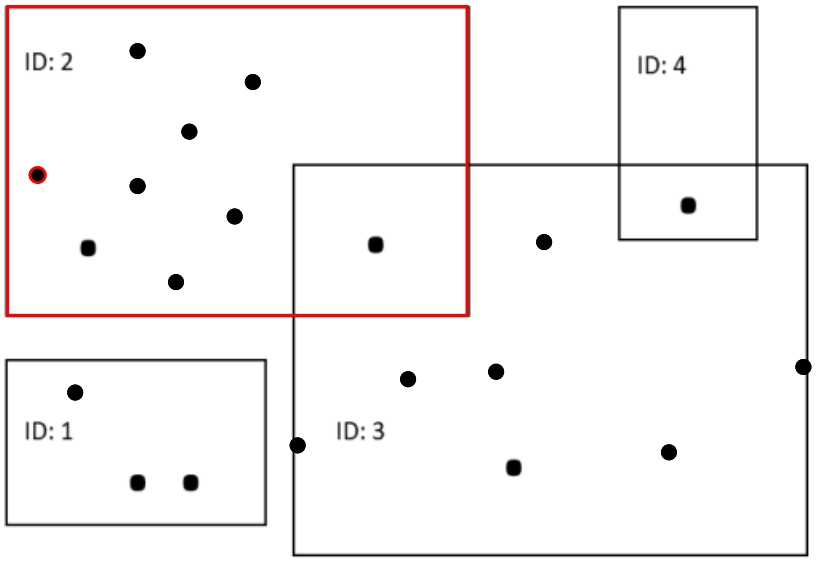
2.1 for rectID= 1 -> number of rectangles

for rectID =2

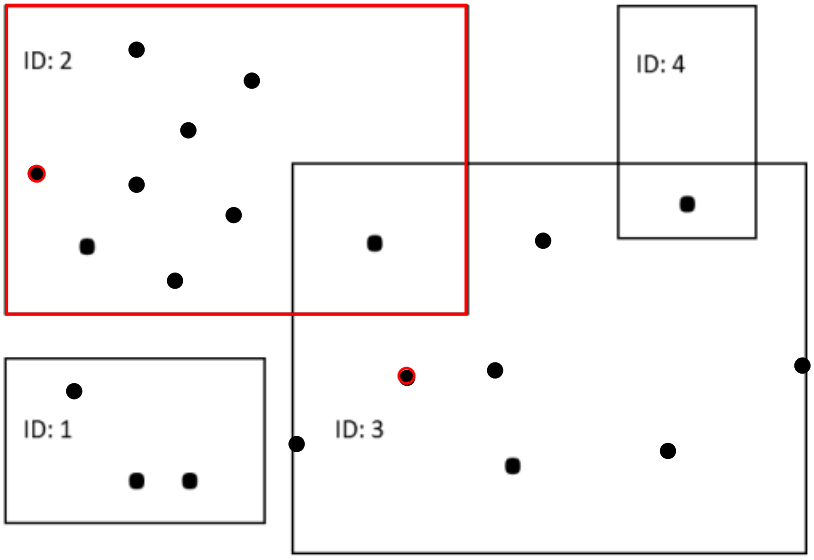
➔first initialize the count to zero



2.1.1 Binary Search on the points to get the lowest x that fits in the rectangle or located on its left edge.

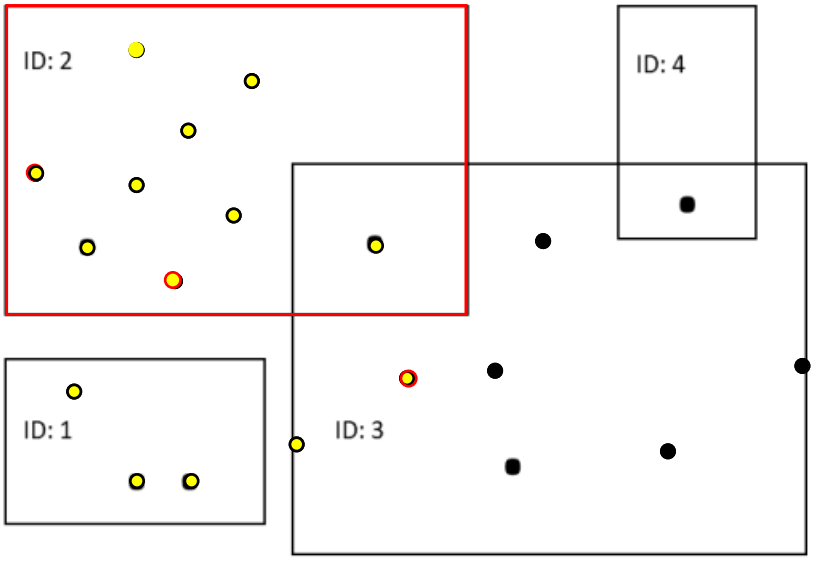


2.1.2 Binary Search on the points (starting from the lower bound) to get the highest x that fits in the rectangle or located on its right edge.

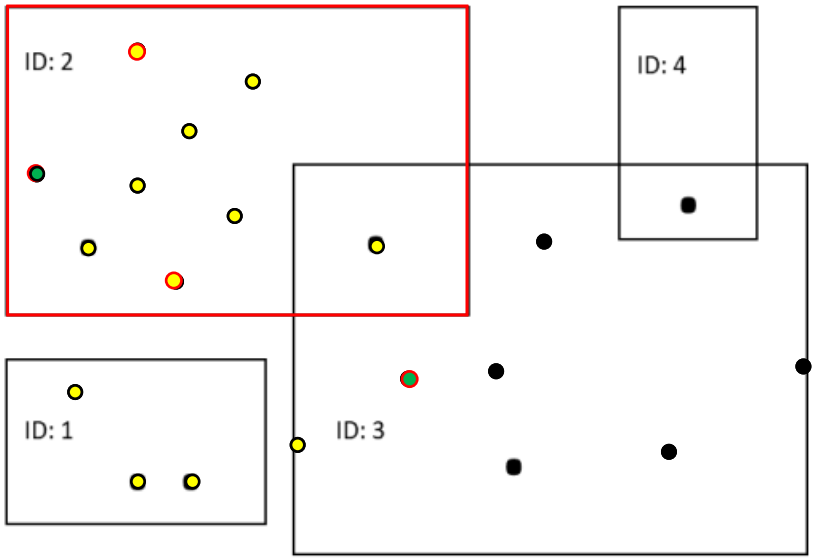


2.1.3 Sort the range of points that fits in the rectangle with respect to x-coordinate.

2.1.4 Binary Search on the to get the lowest y that fits in the rectangle or located on its left edge.



2.1.5 Binary Search on the points (starting from the lower bound) to get the highest x that fits in the rectangle or located on its right edge.



2.1.6 Count = (yUpper -yLower) +1

Print the rectID, count of points included in the output file.

Comparison between Algorithms

<table>
<tr><th></th><th>1st algorithm</th><th>2nd algorithm</th></tr>
<tr><td>Time</td><td>

Very High.
Tremendously affected by both the number of rectangles & the number of points.

<table>
<tr><td>Data_1</td><td>0.049 msec</td></tr>
<tr><td>Data_2</td><td>467.246 msec</td></tr>
<tr><td>Data_3</td><td>434.644 msec</td></tr>
<tr><td>Data_4</td><td>122.437 msec</td></tr>
<tr><td>Data_5</td><td>-</td></tr>
</table>
</td><td>

Better time (lower than the 1st algorithm).
Tremendously affected by the number of rectangles but slightly affected by the number of points.

<table>
<tr><td>Data_1</td><td>0.017 msec</td></tr>
<tr><td>Data_2</td><td>102.816 msec</td></tr>
<tr><td>Data_3</td><td>113.741 msec</td></tr>
<tr><td>Data_4</td><td>46.674 msec</td></tr>
<tr><td>Data_5</td><td>Around 12 h for output 1/5 of the output file</td></tr>
</table>
</td></tr>
</table>

This time taken periods depend on the CPU so it may get a bit faster or slower but this just an indication to figure out the difference in the time taken between the 2 algorithms as the 2nd algorithm takes about 1/3 of the time taken by the 1st algorithm.

<table>
<tr><th></th><th>1st algorithm</th><th>2nd algorithm</th></tr>
<tr><td>Memory Usage</td><td>In the stack ➔
map of Rectangles
vector of points
<table><tr><td>Data_1</td><td>1 MB</td></tr><tr><td>Data_2</td><td>88 MB</td></tr><tr><td>Data_3</td><td>5 MB</td></tr><tr><td>Data_4</td><td>2 MB</td></tr></table></td><td>In the stack➔
map of Rectangles
vector of points
vector of points included within the x-axis range (maximum range got)
<table><tr><td>Data_1</td><td>1 MB</td></tr><tr><td>Data_2</td><td>88 MB</td></tr><tr><td>Data_3</td><td>16 MB</td></tr><tr><td>Data_4</td><td>2 MB</td></tr></table>➔affected by the number of points within the x range in some cases (if it’s a wide range)</td></tr>
</table>

Testing Methodology

I used manual testing, depending on modular unit testing by testing every function independently and make sure that it works properly and output the expected results.

I used the 1st dataset for testing with both algorithms using debugging tools and tracing the variables.

➢ 1st algorithm

After sorting the points with respect to x-coordinates, I start testing the conditions of a point that must be satisfied to be included into a rectangle or located on its edges.

It's a very straight forward algorithm that depends on the condition to consider a point included or not. So, it doesn't take too much effort in testing.

➢ 2nd algorithm

Here after sorting the points with respect to x-coordinates, I started by testing each Binary Search function independently & debug to make sure that the values of the lower and upper bound x for every rectangle in the dataset are output properly, then get that range in another vector to be sorted with respect to y-coordinates, afterwards do the same thing with the values of lower & upper bound y.

Afterwards I start running both algorithms and compare the output files.

Notes:

- When I tried my code on the 5th test case it keeps running for too much and output only a portion of the output this is due to the large number of rectangles and that for points.