Pseudocode for the algorithm(s) and analysis of the algorithms is as following:-

"""
If the number of points = n then we can restrict the maximum number of axis parallel lines by (n - 1). Where n - 1 vertical parallel lines to separate n points. Also n - 1 horizontal lines to separate n points. So the maximum number of axis parallel lines needed is (n - 1).

The first heuristics you are asked to implement is the following local-optimization procedure. Start with an arbitrary feasible solution. Try all combinations of two lines from the current feasible solution, and another line. If the removal of the two lines followed by the addition of the other line results in another feasible solution, then proceed and change the current feasible solution. Repeat trying all combinations, until no combination leads to another feasible solution. Such a procedure is used by the meta-heuristic method **Simulated Annealing**.

Time Analysis:-
1. sorting of coordinates will run in O(n log n) time.
2. The simulated annealing algorithm will run in O(n ^ 6) time complexity.

So the total time complexity of this algorithm will be O(n ^ 6).
"""

```
1     def separating_points_by_axis_parallel_lines(number_of_coordinates:
int, coordinates: []):
2         coordinates = sorted(coordinates)
3         vertical_lines, horizontal_lines =
                    simulated_annealing(number_of_coordinates, coordinates)
4         return vertical_lines, horizontal_lines
```

"""
Here we first got an arbitrary feasible solution and have vertical and horizontal lines out of that. Then getting a new optimal solution by removing two lines and adding a third line.
Time Complexity Analysis:-
1. arbitrary_feasible_solution function runs in O(n^2)
2. while loop will run for max n number of time as we can produce new solution by removing
two lines and adding one and maximum line in the arbitrary solution is n - 1. So while loop
can run for max n - 1 times.
a. optimize_feasible_solution function time complexity is O(n ^ 5).
b. vertical_lines and horizontal_lines assignment will be done in O(n) time.
c. other assignments are constant time complexity.
So the total time complexity of the while loop is O(n ^ 6).

So the total time complexity of the function will be O(n ^ 6).
"""

```
1     def simulated_annealing(number_of_coordinates: int, coordinates: []):
2         vertical_lines, horizontal_lines =
             arbitrary_feasible_solution(number_of_coordinates, coordinates)
3         is_feasible = True
4         while is_feasible is True:
5             is_feasible, simulated_vertical_lines,
             simulated_horizontal_lines = optimize_feasible_solution(
                       number_of_coordinates, coordinates, vertical_lines,
                                                        horizontal_lines)
6             vertical_lines = list(simulated_vertical_lines)
7             horizontal_lines = list(simulated_horizontal_lines)

8         return vertical_lines, horizontal_lines


"""
   Here we have n = number of coordinates
   1. For loop is running for n number of times where the check_coincide
function has a time complexity of O(n). So time complexity of the for loop
will be O(n * (2 *n + c1)) where c1 is a constant time complexity of simple
assignments. So for loop time complexity will be O(n^2).
   2. set to list conversion will be O(n) time for both vertical and
horizontal lines each.
   So the total time complexity of the function is O(n ^ 2).
"""


1     def arbitrary_feasible_solution(number_of_coordinates: int,
coordinates: []):
2         vertical_lines = set()
3         horizontal_lines = set()

4         for index in range(0, number_of_coordinates - 1):
5             if coordinates[index][0] < coordinates[index + 1][0]:
6                 vertical_line = (coordinates[index][0] + coordinates[index
                                                              + 1][0]) / 2
7                 vertical_line = check_coincide(vertical_line, coordinates,
                                                                   False)
8                 vertical_lines.add(vertical_line)
9             elif coordinates[index][0] == coordinates[index + 1][0]:
10                horizontal_line = (coordinates[index][1] +
                                         coordinates[index + 1][1]) / 2
11                horizontal_line = check_coincide(horizontal_line,
                                                        coordinates, True)
12                horizontal_lines.add(horizontal_line)

13        return list(vertical_lines), list(horizontal_lines)




"""
```

Here we are removing two lines in pairs (vertical, vertical), (vertical, horizontal), (horizontal, horizontal) and adding a third line. Third Line can be vertical or horizontal and checking solution feasibility. If a solution is feasible then we are adding solution to the solution space as done in simulated annealing.

1. first nested for loop run for number of vertical lines - 1 which can we max number of coordinates
    a. inner for loop run for number of vertical lines which can we max number of coordinates
        i. two_line_removed_vertical_lines assignment and removal or values is done in O(n) each
        ii. get_un_separated_coordinates function time complexity is O(n ^ 2)
        iii. third_line_simulated_annealing function time complexity is O(n ^ 3)
        iv. vertical_lines and horizontal lines assignment is O(n) each
        v. rest of the operations are constat time complexity
        So total time complexity of the inner for loop is O(n ^4)
    So total time complexity of the inner outer loop is O(n ^5)
2. Similar to first nested while loop second and third while loop complexity will be O(n ^ 5).
3. All other operations will be constant time complexity.

So Total Time complexity of optimize_feasible_solution function is O(n ^ 5).
"""

```
1     def optimize_feasible_solution(number_of_coordinates: int,
            coordinates: [], vertical_lines: [], horizontal_lines: []):
2         for i in range(len(vertical_lines) - 1):
3             for j in range(i + 1, len(vertical_lines)):
4                 two_line_removed_vertical_lines = list(vertical_lines)
5                 two_line_removed_vertical_lines.remove(vertical_lines[i])
6                 two_line_removed_vertical_lines.remove(vertical_lines[j])

7                 un_separated_coordinates =
      get_un_separated_coordinates(number_of_coordinates, coordinates,
      two_line_removed_vertical_lines, horizontal_lines)

8                 is_feasible, simulated_vertical_lines,
                                        simulated_horizontal_lines = \
                third_line_simulated_annealing(number_of_coordinates,
                          coordinates, two_line_removed_vertical_lines,
                                                  horizontal_lines,
                                              un_separated_coordinates)

9                 if is_feasible:
10                    return is_feasible, simulated_vertical_lines,
                                                    simulated_horizontal_lines

11        for i in range(len(vertical_lines)):
```

```
12          for j in range(len(horizontal_lines)):
13              one_line_removed_vertical_lines = list(vertical_lines)
14              one_line_removed_vertical_lines.remove(vertical_lines[i])
15              one_line_removed_horizontal_lines = list(horizontal_lines)
16              one_line_removed_horizontal_lines.remove
                                                    (horizontal_lines[j])

17              un_separated_coordinates =
    get_un_separated_coordinates(number_of_coordinates, coordinates,
                                        one_line_removed_vertical_lines,
                                    one_line_removed_horizontal_lines)
18              is_feasible, simulated_vertical_lines,
                                    simulated_horizontal_lines = \
            third_line_simulated_annealing(number_of_coordinates,
                        coordinates, one_line_removed_vertical_lines,
                                    one_line_removed_horizontal_lines,
                                                un_separated_coordinates)

19              if is_feasible:
20                  return is_feasible, simulated_vertical_lines,
                                                simulated_horizontal_lines

21      for i in range(len(horizontal_lines) - 1):
22          for j in range(i + 1, len(horizontal_lines)):
23              two_line_removed_horizontal_lines = list(horizontal_lines)
24              two_line_removed_horizontal_lines.remove(
                                                    horizontal_lines[i])
25              two_line_removed_horizontal_lines.remove(
                                                    horizontal_lines[j])

26              un_separated_coordinates =
          get_un_separated_coordinates(number_of_coordinates,
                                                coordinates,
                                                vertical_lines,
                                    two_line_removed_horizontal_lines)
27              is_feasible, simulated_vertical_lines,
                                    simulated_horizontal_lines = \
                  third_line_simulated_annealing(number_of_coordinates,
                                        coordinates, vertical_lines,
                                    two_line_removed_horizontal_lines,
                                    un_separated_coordinates)

28              if is_feasible:
29                  return is_feasible, simulated_vertical_lines,
                                                simulated_horizontal_lines

30      return False, vertical_lines, horizontal_lines


"""
   third_line_simulated_annealing function is used to for adding third line
   combination to the algorithm and then checking their feasibility. If
feasible
```

```
   Solution found then returning that to the simulated annealing method to
update
   current feasible solution to better optimized feasible solution.

   Time Complexity Analysis:-
   Maximum un separated coordinates when removing two lines will be lesser
than n
   where n is the number of coordinates.
       1. sorting of un_separated_coordinates will be O(n log n)
       2.  first for loop is used for checking every possible vertical line
between two coordinates
           as coordinates are sorted on x axis so we just have to put
vertical line between two coordinate
           in the same order. Taking a combination of every two coordinates
is not required.
           a. check_coincide function time complexity is O(n).
           b. assignment of simulated_vertical_lines is O(n).
           c. insert_value_in_axis_parallel_lines function time complexity is
O(n).
           d. check_feasibility function time complexity is O(n ^2).
           e. rest of the operations in for loop is constant time.

       So total time complexity of the for loop will be O(n * (n + n + n + n
^2 + c1)) which is
       equivalent to O(n^3).
       3. sorting of un_separated_coordinates will be done on y axis in O(n
log n)
       4.  second for loop is used for checking every possible horizontal
line between two coordinates
           as coordinates are sorted on y axis so we just have to put
horizontal line between two coordinate
           in the same order. Taking a combination of every two coordinates
is not required.
           a. check_coincide function time complexity is O(n).
           b. assignment of simulated_horizontal_lines is O(n).
           c. insert_value_in_axis_parallel_lines function time complexity is
O(n).
           d. check_feasibility function time complexity is O(n ^2).
           e. rest of the operations in for loop is constant time.

       So total time complexity of the for loop will be O(n * (n + n + n + n
^2 + c1)) which is
       equivalent to O(n^3).
       5. Rest of the operations are constant time

   So total time complexity of the function third_line_simulated_annealing
is as following:-
   = O(n log n) + O(n ^3) + O(n log n) + O(n ^3 + O(c1)
   = O(n^3)
"""


1      def third_line_simulated_annealing(number_of_coordinates: int,
```

```
                                **coordinates: [],**
                                **vertical_lines: [],**
                                **horizontal_lines: [],**
                                **un_separated_coordinates: []):**
2         un_separated_coordinates = sorted(un_separated_coordinates)
3         for index in range(len(un_separated_coordinates) - 1):
4             first_coordinate = un_separated_coordinates[index]
5             second_coordinate = un_separated_coordinates[index + 1]

        # Adding third vertical line to the un_separated_coordinates
6             third_vertical_line = (first_coordinate[0] +
                                              second_coordinate[0]) / 2
7             third_vertical_line = check_coincide(third_vertical_line,
                                                   coordinates, False)

8             simulated_vertical_lines = list(vertical_lines)
9             insert_value_in_axis_parallel_lines(simulated_vertical_lines,
                                                   third_vertical_line)
10            is_vertical_simulation_feasible =
                  check_feasibility(number_of_coordinates, coordinates,
                                            simulated_vertical_lines,
                                                   horizontal_lines)

11            if is_vertical_simulation_feasible is True:
12                return True, simulated_vertical_lines, horizontal_lines

13        un_separated_coordinates = sorted(un_separated_coordinates,
                                                   key=lambda x: x[1])
14        for index in range(len(un_separated_coordinates) - 1):
15            first_coordinate = un_separated_coordinates[index]
16            second_coordinate = un_separated_coordinates[index + 1]

              # Adding third horizontal line to the un_separated_coordinates
17            third_horizontal_line = (first_coordinate[1] +
                                              second_coordinate[1]) / 2
18            third_horizontal_line = check_coincide(third_horizontal_line,
                                                   coordinates, True)
19            simulated_horizontal_lines = list(horizontal_lines)
20            insert_value_in_axis_parallel_lines(
                          simulated_horizontal_lines, third_horizontal_line)

21            is_horizontal_simulation_feasible =
                  check_feasibility(number_of_coordinates, coordinates,
                  vertical_lines,
                  simulated_horizontal_lines)

22            if is_horizontal_simulation_feasible is True:
23                return True, vertical_lines, simulated_horizontal_lines

24        return False, vertical_lines, horizontal_lines

"""
```

This function is Used to insert elements in a sorted list of axis parallel lines.

Time complexity of the function will be O(n) as the maximum number of axis parallel lines for the solution is n - 1. Here for loops is running for the number of axis parallel lines.

insertion at index n will also be O(n) time.
Other operations are constant time in the algorithm.

So total time complexity of the function will be O(2 * n) which is equivalent to O(n).

```
"""
1     def insert_value_in_axis_parallel_lines(axis_parallel_lines, value):
2         index = len(axis_parallel_lines)
3         for i in range(len(axis_parallel_lines)):
4             if axis_parallel_lines[i] > value:
5                 index = i
6                 break

7         axis_parallel_lines.insert(index, value)
8         return axis_parallel_lines


"""
```

This function is used for checking if axis parallel lines coincide on coordinates. And if it coincides with any point then we are increasing the axis parallel line value to 0.25. Axis Parallel Line can be either horizontal or vertical depending on is_horizontal argument.

Time Complexity:
For loop runs for O(n) times where n = number of coordinates. All other assignments inside are constant time. So the total time complexity of this function will be O(n).

```
"""
1     def check_coincide(axis_parallel_line: int, coordinates: [],
                                                    is_horizontal: bool):
2         for coordinate in coordinates:
3             if is_horizontal is False and axis_parallel_line ==
                                                        coordinate[0]:
4                 axis_parallel_line += 0.25
5             if is_horizontal is True and axis_parallel_line ==
                                                        coordinate[1]:
6                 axis_parallel_line += 0.25

7         return axis_parallel_line


"""
```

Method is used to check if separating n points using axis parallel lines is feasible solution or not.
- axis parallel lines are represented by vertical_lines and horizontal_lines

```
   Logic:-
       Using both x_axis_separation_map and y_axis_separation_map checking
if there is any combination between two coordinates which is yet not
separated. If a combination is present like that then the solution is not
feasible.

   Total time complexity is as following:-
       1. The time complexity of the generate_separation_maps function is
O(n ^ 2).
       2. nested for loop have inner for loop and both run for n = number
of coordinates. So time
       complexity of the nested for loop will be O(n ^2 ).
       3. un_separated_coordinates set to list conversion is O(n).

   So the total time complexity of the function is O(n ^ 2).
"""
1      def check_feasibility(number_of_coordinates: int, coordinates: [],
                             vertical_lines: [], horizontal_lines: []):
2          x_axis_separation_map, y_axis_separation_map, replaced_index =
                                                  generate_separation_maps \
                                      (number_of_coordinates, coordinates,
                                           vertical_lines, horizontal_lines)

3          for i in range(0, number_of_coordinates):
4              for j in range(i + 1, number_of_coordinates):
5                  i_replaced_index = replaced_index[i]
6                  j_replaced_index = replaced_index[j]
7                  if x_axis_separation_map[i][j] is False and \
       (y_axis_separation_map[i_replaced_index][j_replaced_index] is True or
       y_axis_separation_map[j_replaced_index][i_replaced_index] is True):
8                      x_axis_separation_map[i][j] = True

9                  if x_axis_separation_map[i][j] is False:
10                     return False

11         return True


"""
   Method is used for getting not separated points by axis parallel lines.
   Total time complexity is as following:-
       1. The time complexity of the generate_separation_maps function is O(n
^ 2).
       2. nested for loop have inner for loop and both run for n = number of
coordinates. So time
       complexity of the nested for loop will be O(n ^2 ).
       3. un_separated_coordinates set to list conversion is O(n).

   So the total time complexity of the function is O(n ^ 2).
"""
1      def get_un_separated_coordinates(number_of_coordinates: int,
             coordinates: [], vertical_lines: [], horizontal_lines: []):
2          x_axis_separation_map, y_axis_separation_map, replaced_index =
```

```
                                                    generate_separation_maps \
                                            (number_of_coordinates, coordinates,
                                                 vertical_lines, horizontal_lines)
3         un_separated_coordinates = set()
4         for i in range(0, number_of_coordinates):
5             for j in range(i + 1, number_of_coordinates):
6                 i_replaced_index = replaced_index[i]
7                 j_replaced_index = replaced_index[j]
8                 if x_axis_separation_map[i][j] is False and \
       (y_axis_separation_map[i_replaced_index][j_replaced_index] is True or
       y_axis_separation_map[j_replaced_index][i_replaced_index] is True):
9                     x_axis_separation_map[i][j] = True

10                if x_axis_separation_map[i][j] is False:
11                    un_separated_coordinates.add(coordinates[i])
12                    un_separated_coordinates.add(coordinates[j])
13        return list(un_separated_coordinates)


"""
   Following function is used to prepare vertical lines separation map and
horizontal line separation map.
   Separation map is 2d array which denotes combination of two coordinate and
if value is true then it is
   separated by a line.

   Logic:-
       First sort coordinates on the basis of x axis then updating
x_axis_separation_map
       which update separation relation between two coordinates.
           - As points are sorted on x axis if first_x_axis coordinate and
second_x_axis coordinate
           is separated using vertical line then we can say first_x_axis
coordinate is separated
           to bigger_x_axis coordinates.

       With same logic as above now sorting coordinates on y_axis and
producing y_axis_separation_map

   Time Complexity Analysis:-
   n = number of coordinates
   1. x_axis_sorted_coordinates is sorted in O(n log n) time
   2. x_axis_separation_map initialization is O(2 * n ^ 2) time
       a. O(n ^ 2) time to initiate value to None
       b. O(n ^ 2) time to update value as True

   3.After x_axis_separation_map initialization we update the coordinates
combination which are
   not separated by vertical lines
       a. first we see separation between i and i + 1 coordinates in
x_axis_sorted_coordinates
       this step take O(n) time as there can only be max n - 1 number of
vertical lines
```

```
       b. if there is no separation then need to update previous coordinates
eg.
           let's say if we have
           i. 2, 3 coordinate as not separated
           ii. we found out that 3, 4 coordinate is also not separated
           iii. now we need to update that 2, 4 is also not separated
       This step take O(n) time as there can be max n - 1 coordinate before
current coordinate.
   This step in total take time complexity of O(n ^2) as outer loop also runs
n times.

   4. This step will be similar to step 1, 2, 3 combined but we will get
y_axis_separation_map
   Time complexity will be equivalent to step 1, 2, 3

   5. find_replaced_index run for O(n ^ 2) time to find new index after
sorting coordinates on y_axis

   So the total Time complexity of this function is O(n ^ 2).
"""


1      def generate_separation_maps(number_of_coordinates: int, coordinates:
                           [], vertical_lines: [], horizontal_lines: []):
2          x_axis_sorted_coordinates = sorted(coordinates)

   # assume all coordinates are separated at first
   # is first line and second line is divided then first line is divided to
   # every other line so update all combination with first coordinate as
   # separated

3          x_axis_separation_map = [[None for _ in range(0,
       number_of_coordinates)] for _ in range(0, number_of_coordinates)]
4          for i in range(0, number_of_coordinates):
5              for j in range(i + 1, number_of_coordinates):
6                  x_axis_separation_map[i][j] = True

7          for i in range(0, number_of_coordinates - 1):
8              first_x_axis = x_axis_sorted_coordinates[i][0]
9              second_x_axis = x_axis_sorted_coordinates[i + 1][0]

10             separation = False
11             for vertical_line in vertical_lines:
12                 if first_x_axis < vertical_line < second_x_axis:
13                     separation = True
14                     break

15             x_axis_separation_map[i][i + 1] = separation

16             if separation is False:
17                 for j in range(i - 1, -1, -1):
18                     if x_axis_separation_map[j][j + 1] is False:
```

```
                    x_axis_separation_map[j][i + 1] = False
                else:
                    break

    y_axis_separation_map = [[None for _ in range(0,
number_of_coordinates)] for _ in range(0, number_of_coordinates)]
    for i in range(0, number_of_coordinates):
        for j in range(i + 1, number_of_coordinates):
            y_axis_separation_map[i][j] = True


    y_axis_sorted_coordinates = sorted(coordinates, key=lambda x:
                                                                x[1])
    for i in range(0, number_of_coordinates - 1):
        first_y_axis = y_axis_sorted_coordinates[i][1]
        second_y_axis = y_axis_sorted_coordinates[i + 1][1]

        separation = False
        for horizontal_line in horizontal_lines:
            if first_y_axis < horizontal_line < second_y_axis:
                separation = True
                break

        y_axis_separation_map[i][i + 1] = separation

        if separation is False:
            for j in range(i - 1, -1, -1):
                if y_axis_separation_map[j][j + 1] is False:
                    y_axis_separation_map[j][i + 1] = False
                else:
                    break


    replaced_index = find_replaced_index(number_of_coordinates,
             x_axis_sorted_coordinates, y_axis_sorted_coordinates)
    return x_axis_separation_map, y_axis_separation_map,
                                                        replaced_index


"""
   To find the replaced indexes we are using two for loops which leads to
time complexity of O(n ^ 2).
   All other things are constant time in this function.
"""


def find_replaced_index(number_of_coordinates: int,
      x_axis_sorted_coordinates: [], y_axis_sorted_coordinates: []):
    replaced_index = [-1 for _ in range(0, number_of_coordinates)]
    for x_axis_sorted_coordinate_index in range(0,
                                            number_of_coordinates):
        for y_axis_sorted_coordinate_index in range(0,
                                            number_of_coordinates):
```

```
5               if x_axis_sorted_coordinates[
                            x_axis_sorted_coordinate_index] == \
                y_axis_sorted_coordinates[y_axis_sorted_coordinate_index]:
6                   replaced_index[x_axis_sorted_coordinate_index] =
                                        y_axis_sorted_coordinate_index

7       return replaced_index
```

One Instance of an algorithm on which the algorithm fails to return the optimum solution:-

I ran my algorithm on provided instances with the following instance my algorithm given output of minimum 14 axis parallel lines.

```
30
1 7
2 28
3 4
4 13
5 14
6 26
7 21
8 5
9 25
10 6
11 19
12 30
13 17
14 18
15 3
16 27
17 11
18 15
19 24
20 1
21 2
22 10
23 12
24 29
25 8
26 16
27 20
28 23
29 9
30 22
```

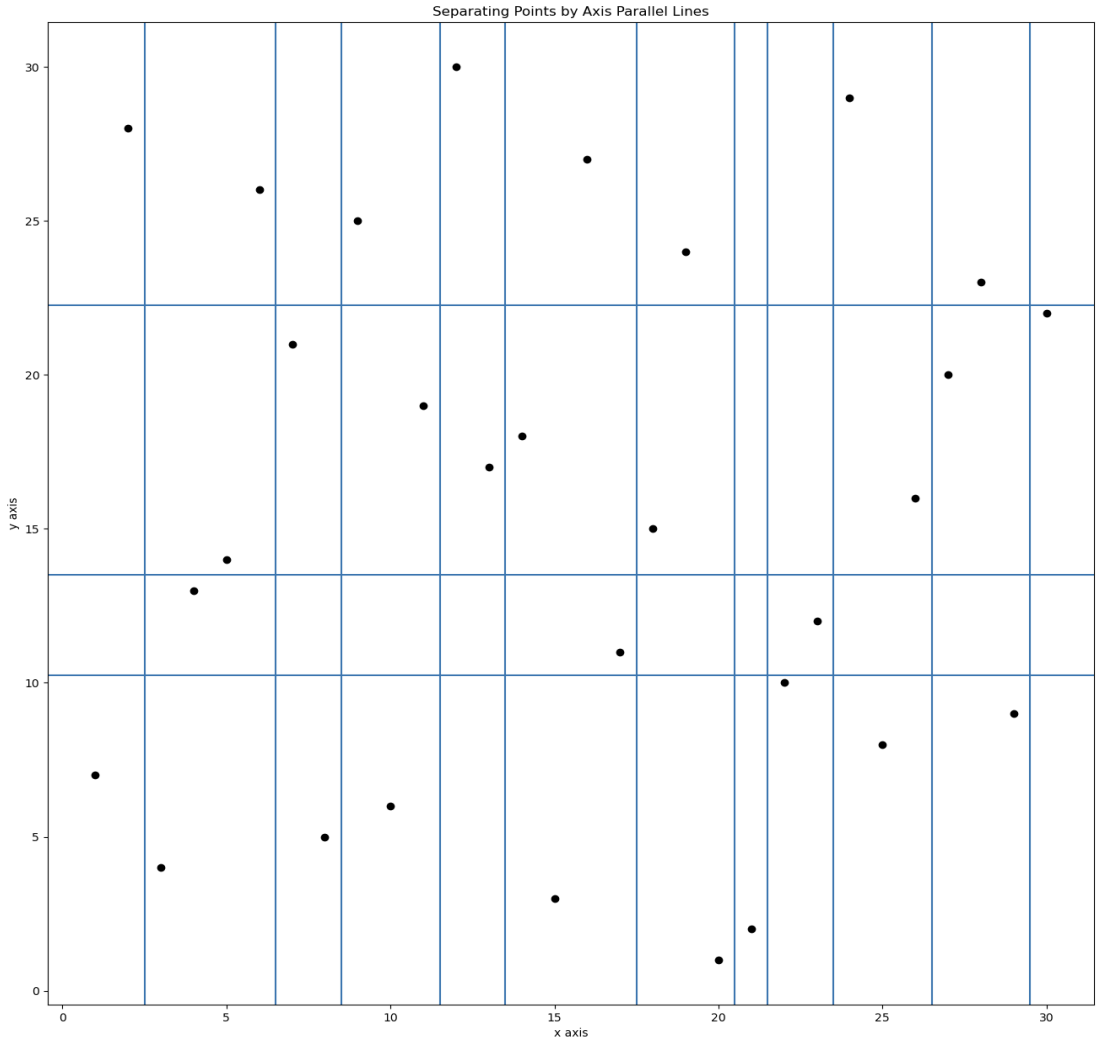
Run of the algorithm on the instance:-

The output produced by algorithm is as following:-

```
14
h 10.25
```

```
h 13.5
h 22.25
v 2.5
v 6.5
v 8.5
v 11.5
v 13.5
v 17.5
v 20.5
v 21.5
v 23.5
v 26.5
v 29.5
```

Visualized Output is as following:-



Better Solution:-

As the professor provided a better solution for the above instance my algorithm failed to produce 13 axis parallel lines to solve this problem. So better solution is as following:-

13
h 14.5
h 22.5
h 5.5
v 22.5
v 7.5
v 26.5
v 11.5
h 10.5
v 4.5
v 13.5
v 16.5
v 20.5
v 27.5

Visualized Output is as following:-