**Separating n-points using Greedy approach:**

***1. Pseudocode:***

SEPARATE\_POINTS\_GREEDY( )

//OPERATION: Separate n-points in a graph with axis-parallel lines using Greedy approach.

//INPUT: The program will read instances files each containing the co- ordinates of the points.

The points are sorted along x-axis and they won't share each other's axes.

//OUPUT: A Feasible solution containing the set 'S' of axis-parallel lines that separate the given instance points such that |S| is minimum. The solutions are printed in a file.

1. //Integer identifiers
2. file\_inst = 1
3. inst\_count = 0
4. x\_lines[] = 0
5. y\_lines[] = 0
6. max\_files = 100
7. //Structure identifiers
8. st\_points = 0
9. st\_lines = 0
10. while file\_inst < max\_files
11. READ\_FILE(file\_inst, st\_points)
12. //Sort x and y axes values of the co-ordinates.
13. QSORT\_POINTS(st\_points.x[0], st\_points.y[0])
14. MAKE\_CONNECTIONS(st\_points)
16. //Finding axis parallel lines for both the axis, X = 0 and Y = 1
17. FIND\_AXIS\_PARALLEL\_LINES(0, st\_points, st\_lines)
18. FIND\_AXIS\_PARALLEL\_LINES(1, st\_points, st\_lines)
19. while i < st\_lines.x\_total and j < st\_lines.y\_total
20. conxn\_status = CHECK\_LINE\_CONNECTIONS(st\_lines.x[i], st\_points)
21. if conxn\_status == 1
22. COMMIT\_LINE(st\_lines.x[i], st\_points, st\_lines)
23. endif
24. i = i + 1
25. conxn\_status = CHECK\_LINE\_CONNECTIONS(st\_lines.y[j], st\_points)
26. if conxn\_status == 1
27. COMMIT\_LINE(st\_lines.y[j], st\_points, st\_lines)
28. endif
29. j = j + 1
30. endwhile
32. SAVE\_SOLUTION\_IN\_FILE(inst\_count)
33. file\_inst = file\_inst + 1
34. endwhile
35. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

MAKE\_CONNECTIONS(st\_points)

//OPERATION: Form connections between all the points/co-ordinates in the graph.

//INPUT: st\_points - structure having information on points that are sorted along their respective axis

//OUPUT: The structure "st\_points" updated with connection information of the co- ordinates

1. i = 0
2. j = 0
3. while i < st\_points.total
4. st\_points.coord[i].index = i
5. while j < st\_points.total
6. if i != j
7. st\_points.coord[i].connection[j] = address\_of(st\_points.coord[j])
8. st\_points.coord[i].connection\_count = st\_points.coord[i].connection\_count + 1
9. else
10. st\_points.coord[i].connection[j] = NULL
11. endif
12. endwhile
13. endwhile
14. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

FIND\_AXIS\_PARALLEL\_LINES(axis, st\_points, st\_lines)

//OPERATION: Recursively divide X and Y axis into half, as a line in the middle of a graph disconnects most of the points.

//INPUT: Structure "st\_points" having points and connection information

//OUTPUT: Structure "st\_lines" having the extracted axis parallel lines of the graph.

1. //Stop condition for recursion
2. if st\_points.begin == st\_points.total
3. return
4. endif
5. //Calculate the half range
6. range = st\_points.total
7. if range == 0 OR range == 1
8. return
9. else if range%2
10. middle = (range-1) / 2
11. else
12. middle = range / 2
13. endif
14. if axis == 0
15. axis\_points = st\_points.x[0]
16. else
17. axis\_points = st\_points.y[0]
18. endif
19. //Find the intercept and update that in the line information
20. if middle != 0
21. point1 = axis\_points[ st\_points.begin + middle ]
22. point2 = axis\_points[ st\_points.begin + (middle + 1)]
23. length = point2 - point1
24. half\_length = length/2
25. middle\_point = point1 + half\_length
27. if axis == 0
28. UPDATE\_LINES\_INFO(st\_lines.x[0])
29. else
30. UPDATE\_LINES\_INFO(st\_lines.y[0])
31. endif
32. //Recurse on both sides of the middle point
33. if range != 2
34. //From left to middle
35. st\_points.total = st\_points.begin + middle
36. FIND\_AXIS\_PARALLEL\_LINES(0, st\_points, st\_lines)
38. //From middle to end
39. st\_points.begin = st\_points.begin + middle
40. FIND\_AXIS\_PARALLEL\_LINES(0, st\_points, st\_lines)
41. endif
42. endif
43. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CHECK\_LINE\_CONNECTIONS(st\_axis\_line, st\_points)

//OPERATION: Check for connection(edge) between two points

//INPUT: Structure "st\_axis\_line" having the given axis's parallel lines

//OUTPUT: Connection status: 1 = TRUE or 0 = FALSE

1. axis = st\_axis\_line.axis
2. intercept = st\_axis\_line.intercept\_point
3. right\_point = GET\_NEAREST\_RIGHT\_POINT(axis, intercept, st\_points)
4. if axis == 0
5. ptr\_points = st\_points.x[0]
6. else
7. ptr\_points = st\_points.y[0]
8. endif
9. i = 0
10. j = 0
11. while i < (right\_point+1)
12. j = right\_point+1
13. while j < st\_points.total
14. if (st\_points.coord[ptr\_points[i]->index].connection[ptr\_points[j] ->index]) != NULL
15. return 1
16. endif
17. j = j + 1
18. endwhile
19. i = i + 1
20. endwhile
21. return 0
22. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

COMMIT\_LINE(line, st\_points, st\_lines)

//OPERATION: Add the line into the solution and disconnect all the connections which the line crosses

//INPUT: Structure "line" to be added into the solution and "st\_points" having all the points

//OUTPUT: Structure "st\_lines" with the added lines

1. axis = line.axis
2. intercept = line.intercept
3. right\_point = GET\_NEAREST\_RIGHT\_POINT(axis, intercept)
4. //Store the lines that are added to the solution
5. n\_lines = 0
6. st\_lines.added\_lines[n\_lines] = line
7. if axis == 0
8. ptr\_points = st\_points.x[0]
9. else
10. ptr\_points = st\_points.y[0]
11. endif
12. i = 0
13. j = right\_point
14. while i <= right\_point
15. j = right\_point + 1
16. while j < st\_points.total
17. DISCONNECT\_POINTS(ptr\_points[i]->index, ptr\_points[j]->index)
18. j = j + 1
19. endwhile
20. i = i + 1
21. endwhile
22. n\_lines = n\_lines + 1
23. st\_lines.added\_line\_count = n\_lines
25. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

***2. Run-time Analysis:***

The implementation for SEPARATE\_POINTS\_GREEDY( ) is a standalone algorithm that reads co-ordinate values from multiple files, finds their axis parallel lines, separates the given co-ordinates with minimal number of axis parallel lines and stores the result into multiple files.

As given in the problem statement, the input instances would have atmost 100 points and we can have upto 100 instances, having various combinations of the points (sorted along x-axis). We have in **lines 12 to 40** a while-loop that cycles over the given number of instance files. So, at the maximum we have ‘**n**’ time complexity for this loop.

**Line 16** is a sorting operation using Quick-Sort, that, in worst-case, would reach **n^2**.

**Line 18** is a function that does connecting the points of the graph. Its implementation, as given by the lines 61-74, contains two while-loops each for n-points. So, it would be **n^2** complexity for this operation.

**Lines 21 and 22** finds the axis parallel lines of both the axes. The function, which is implement in the lines 85 to 133, is a recursive function, that runs in ‘nlogn’ time. For 2 axes it would be **2nlogn**.

**Lines 24 to 36** has an inner while-loop that runs up to maximum number of the respective co-ordinate axes, which is ‘**n**’ for the total n-points. Within it, it calls the function Commit(), that at maximum runs in **n^2** time. Hence the whole loop of this code section may take upto **n^3** time to run.

Overall, we have:

n(n^2 + n^2 + 2nlogn + n^3) = n(2n^2 + 2nlogn + n^3)

= 2n^3 + 2(n^2)logn + n^4

Omitting lower order terms, we get:

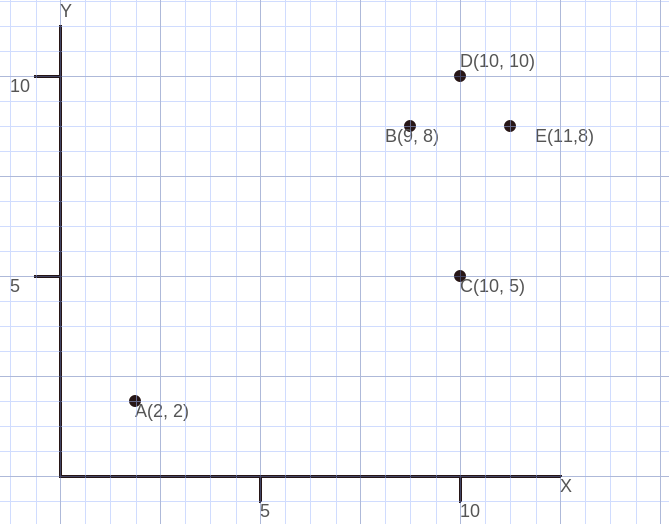
**O(n^4)**, which is the complexity of the algorithm that separates n-points in a graph using greedy approach.

***3. Failure Case of the greedy approach:***

We have assumed in our algorithm that no 2 points share the same ‘x’ or ‘y co-ordinate axis. But, if we alter this assumption the algorithm won’t find an optimal solution, as the greedy approach fails for arbitrary inputs that have significant overlaps.

For example, consider an instance with 5 points in the graph:

A(2, 2), B(9, 8), C(10, 5), D(10, 10), E(11, 8)



We can see that there exists overlaps for the points B & E, and C & D for co-ordinate values y = 8 and x = 10 respectively.

The Greedy method gives the following solution (with 4 lines) :

v 10.0

h 8.0

v 9.5

h 6.5

While an optimum solution for the same would be:

v 10.5

v 9.5

h 6.5

Again, if we alter the same input and ensure that the points have no-overlap co-ordinates, we get an optimal solution. Let the altered points be:

A(2, 2), B(8, 8), C(9, 5), D(10, 10), E(11, 7)

For this input instance, we get:

v 9.5

h 7.5

v 8.5

- which is an optimum solution.

Thus, we can find that the Greedy approach would fail to return an optimum solution if we have inputs of the graph with overlaps.

***References:***

1) Greedy algorightm code: <https://github.com/nickziv/sep_pts>, Author: nickviz

2) Book: Field-Programmable Logic and Applications, by Reiner W. Hartenstein, Andres Keevallik. Page: 131, Section: 2.2, Successive bipartitioning.

2) <http://alg12.wikischolars.columbia.edu/file/view/GREEDY.pdf/305349730/GREEDY.pdf>