Line Sweeping

COMP 4951: Advanced Problem Solving
Micah Stairs

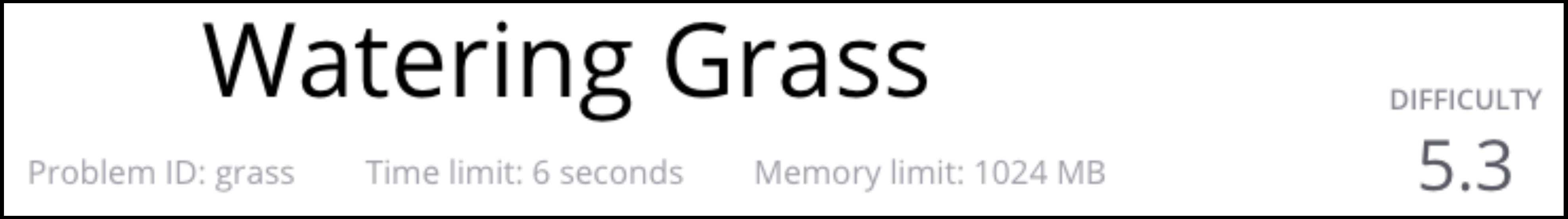
Outline

- Definition
- **Problem #1: Watering Grass**
- **Problem #2: Closest Pair**
- **Problem #3: Arable Area**
- **Problem #4: Rectangle Land**
- Conclusion
- Sources

Definition

- In computational geometry, a **sweep line** algorithm or **plane sweep** algorithm is a type of algorithm that uses a conceptual sweep line or sweep surface to solve various problems in Euclidean space. It is one of the key techniques in computational geometry.

Problem #1: Watering Grass



Watering Grass

Problem ID: grass   Time limit: 6 seconds   Memory limit: 1024 MB

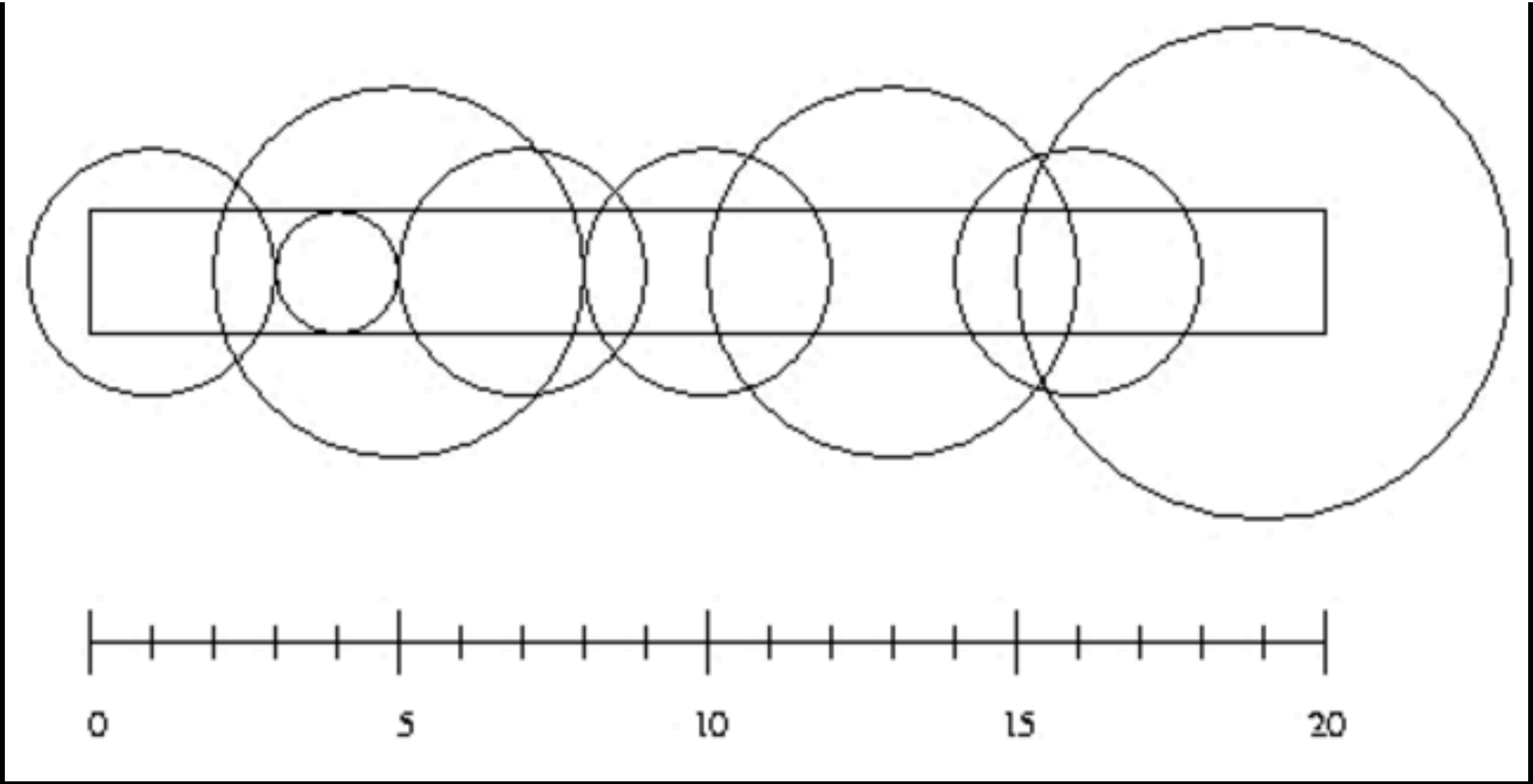
DIFFICULTY

5.3

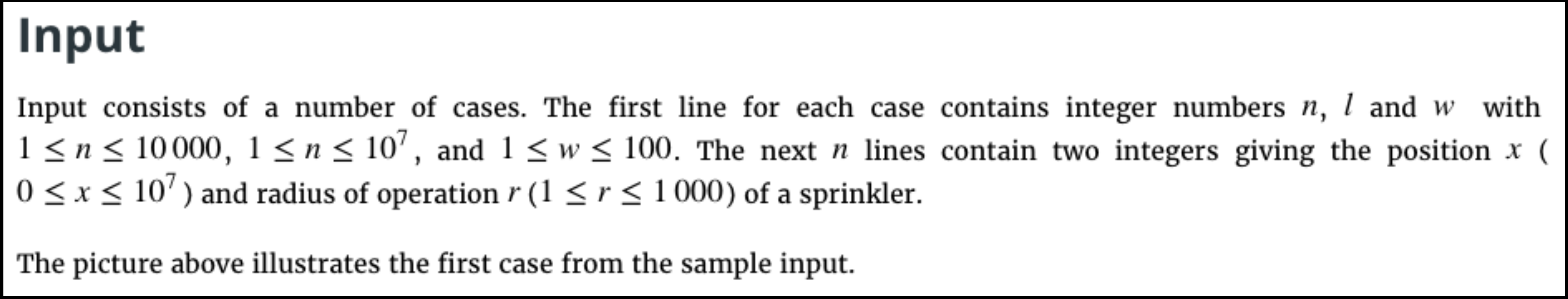
Problem Description

n sprinklers are installed in a horizontal strip of grass l meters long and w meters wide. Each sprinkler is installed at the horizontal center line of the strip. For each sprinkler we are given its position as the distance from the left end of the center line and its radius of operation.

What is the minimum number of sprinklers to turn on in order to water the entire strip of grass?



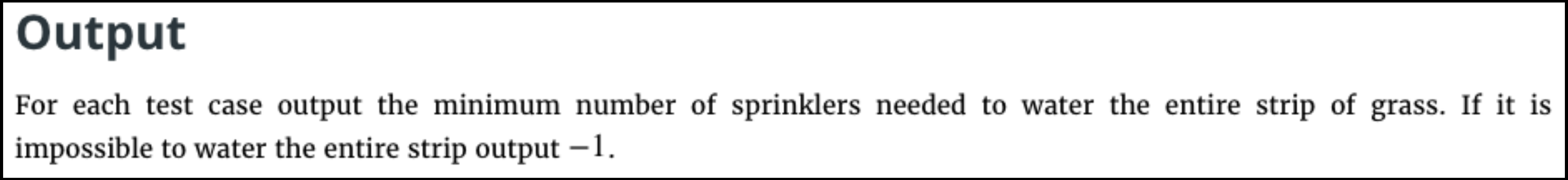
Input and Output



Input

Input consists of a number of cases. The first line for each case contains integer numbers n, l and w with $1 \le n \le 10\,000$, $1 \le n \le 10^7$, and $1 \le w \le 100$. The next n lines contain two integers giving the position x ($0 \le x \le 10^7$) and radius of operation r ($1 \le r \le 1\,000$) of a sprinkler.

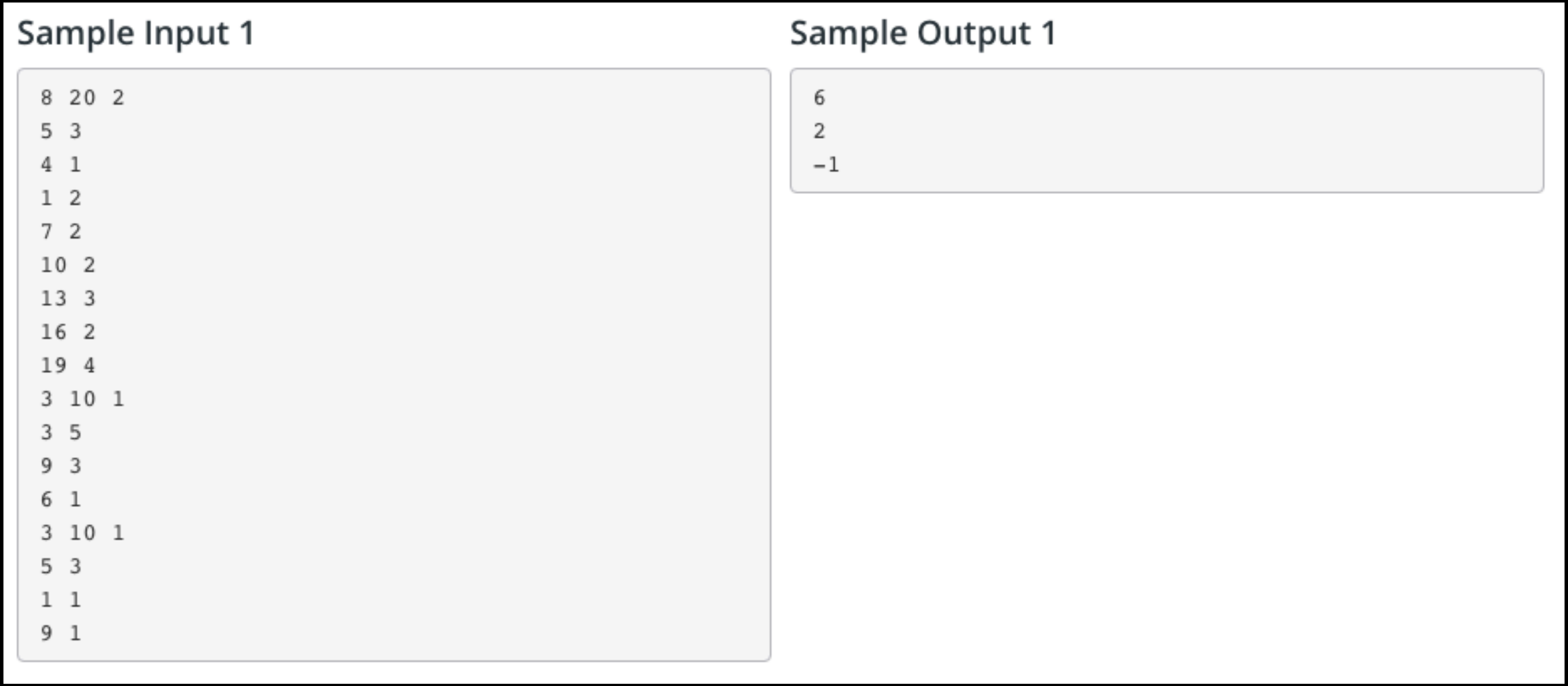
The picture above illustrates the first case from the sample input.



Output

For each test case output the minimum number of sprinklers needed to water the entire strip of grass. If it is impossible to water the entire strip output -1.

Test Data



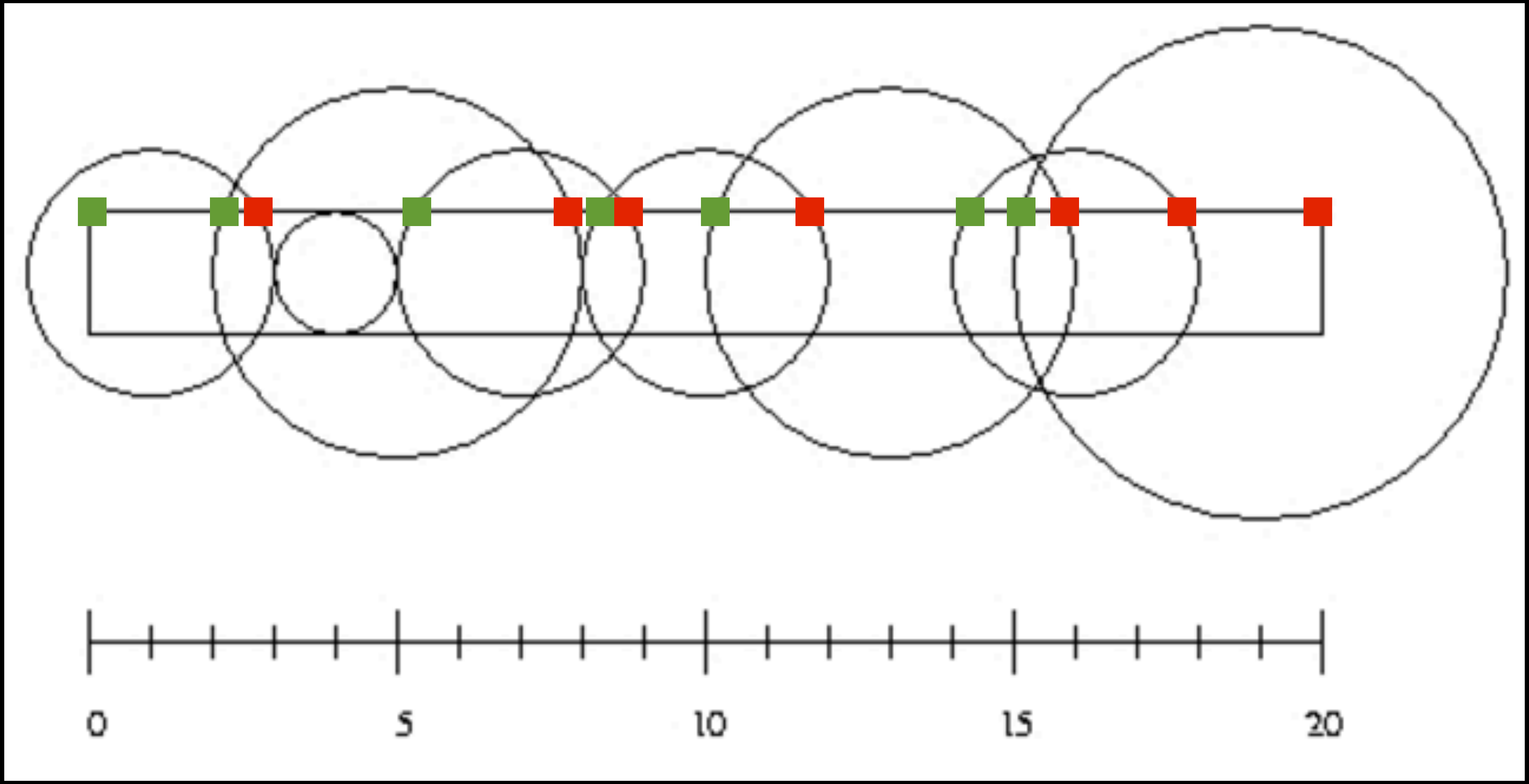
Sample Input 1

```
8 20 2
5 3
4 1
1 2
7 2
10 2
13 3
16 2
19 4
3 10 1
3 5
9 3
6 1
3 10 1
5 3
1 1
9 1
```

Sample Output 1

```
6
2
-1
```

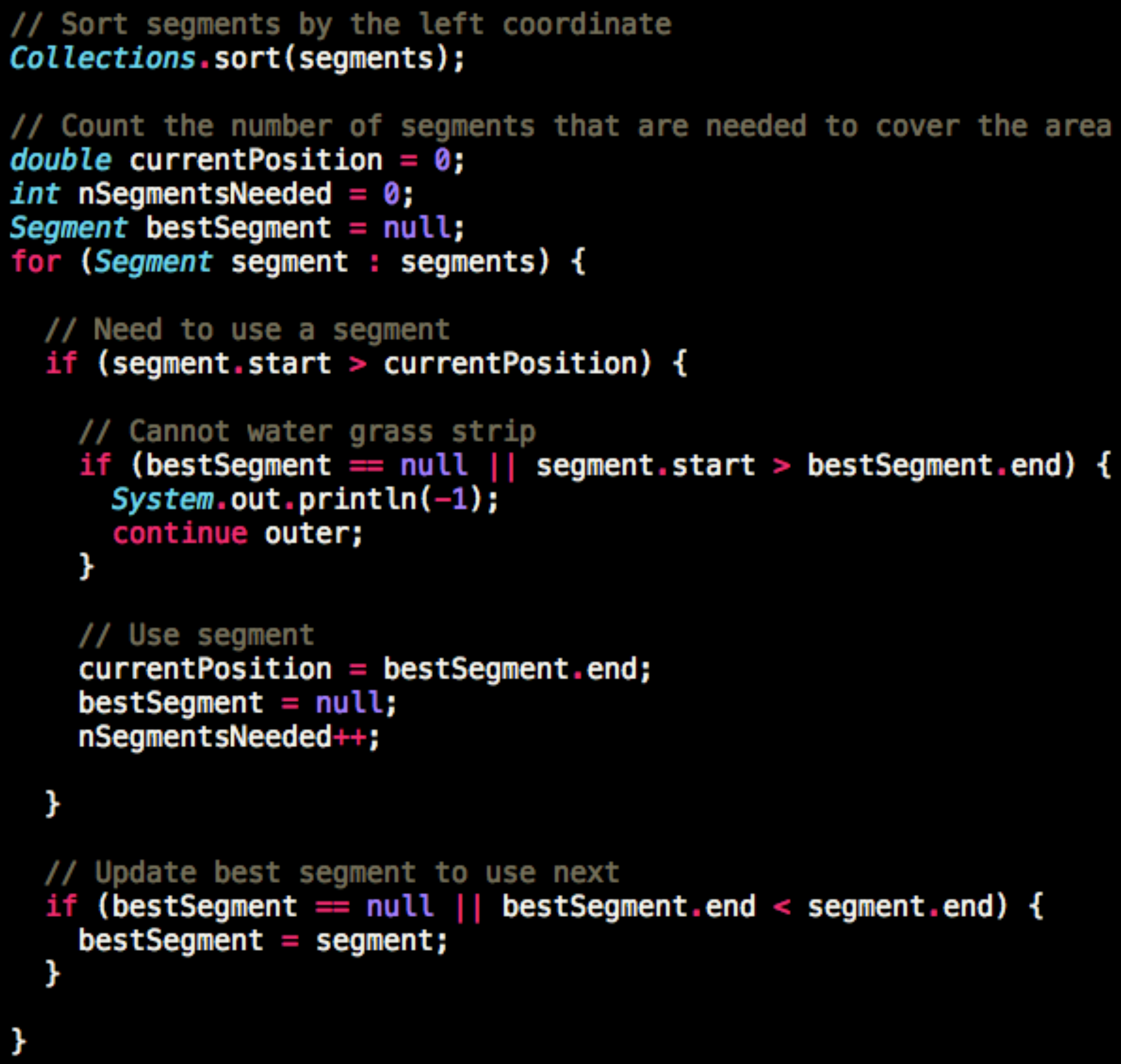
Approach



Solution: O(n log(n))

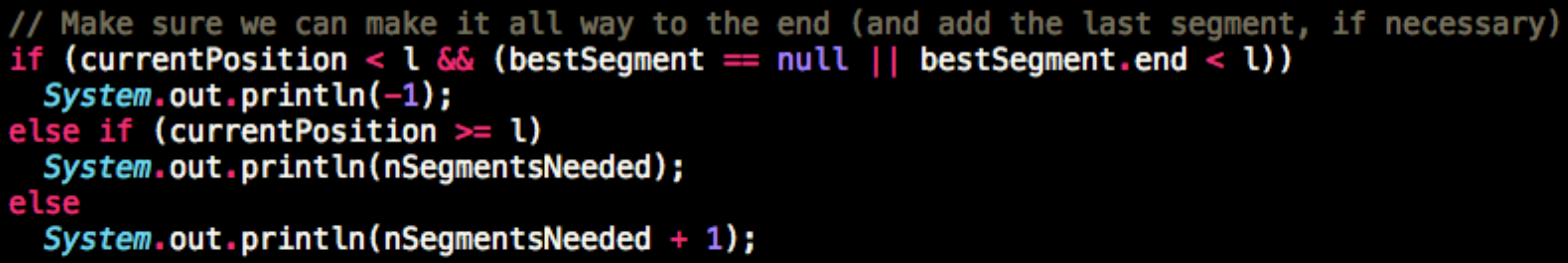
```java
// Read sprinklers and keep list of segments that intersect with the top of strip
List<Segment> segments = new ArrayList<Segment>();
for (int i = 0; i < n; i++) {
  split = br.readLine().split(" ");
  int x = Integer.valueOf(split[0]);
  int r = Integer.valueOf(split[1]);

  // Ignore sprinklers whose radius of operation is too small
  if (r > w/2.0) {

    // Find intersection points with top of strip
    double halfWidth = w/2.0;
    double offset = Math.sqrt(r*r - halfWidth*halfWidth);
    double x1 = x - offset;
    double x2 = x + offset;

    // Trim ends
    x1 = Math.max(0, x1);
    x2 = Math.min(l, x2);

    // Add sprinkler to list
    if (x2 > x1)
      segments.add(new Segment(x1, x2));

  }

}
```

Solution: O(n log(n))



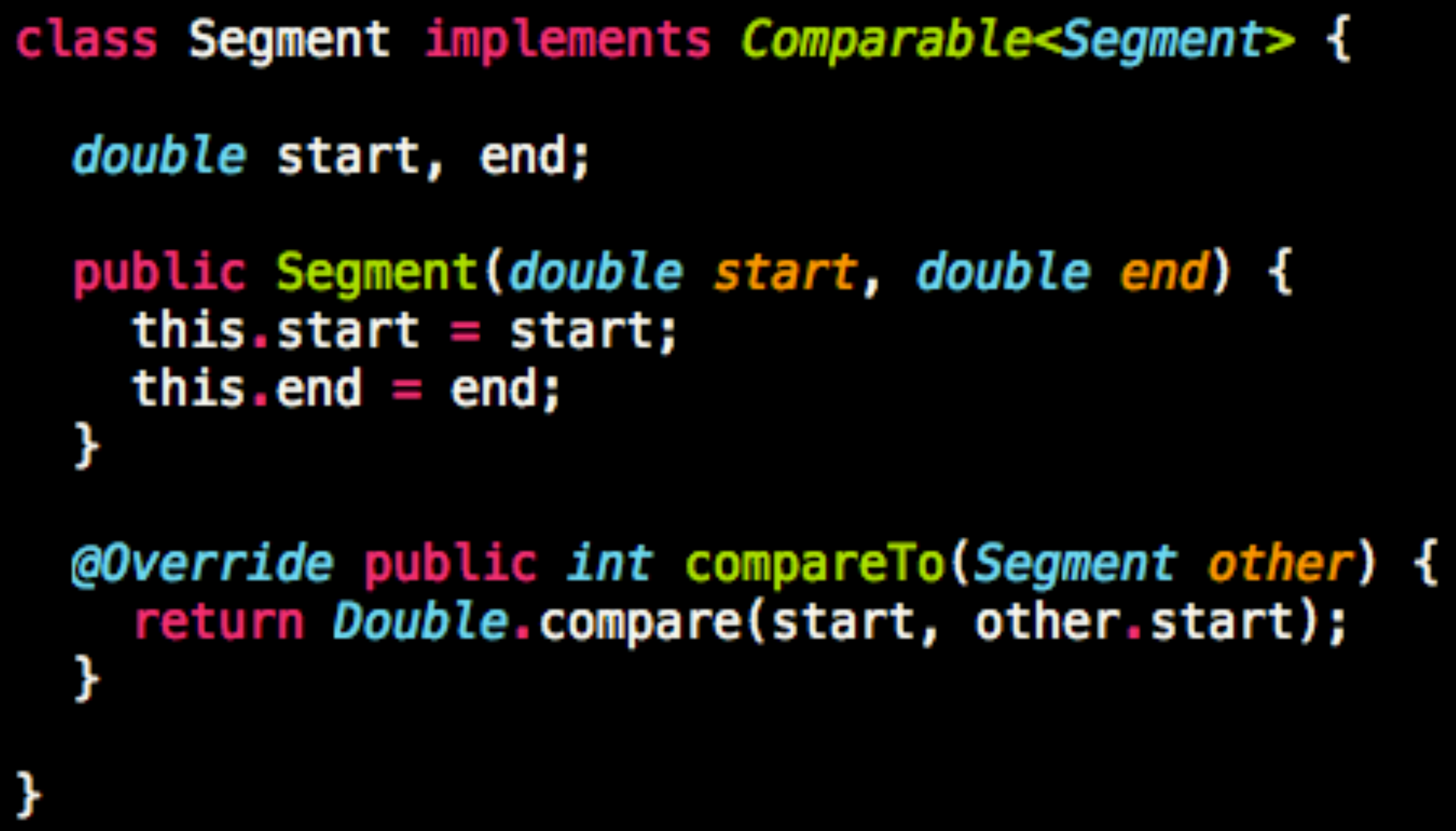
```
// Sort segments by the left coordinate
Collections.sort(segments);

// Count the number of segments that are needed to cover the area
double currentPosition = 0;
int nSegmentsNeeded = 0;
Segment bestSegment = null;
for (Segment segment : segments) {

  // Need to use a segment
  if (segment.start > currentPosition) {

    // Cannot water grass strip
    if (bestSegment == null || segment.start > bestSegment.end) {
      System.out.println(-1);
      continue outer;
    }

    // Use segment
    currentPosition = bestSegment.end;
    bestSegment = null;
    nSegmentsNeeded++;

  }

  // Update best segment to use next
  if (bestSegment == null || bestSegment.end < segment.end) {
    bestSegment = segment;
  }

}
```

Solution: O(n log(n))



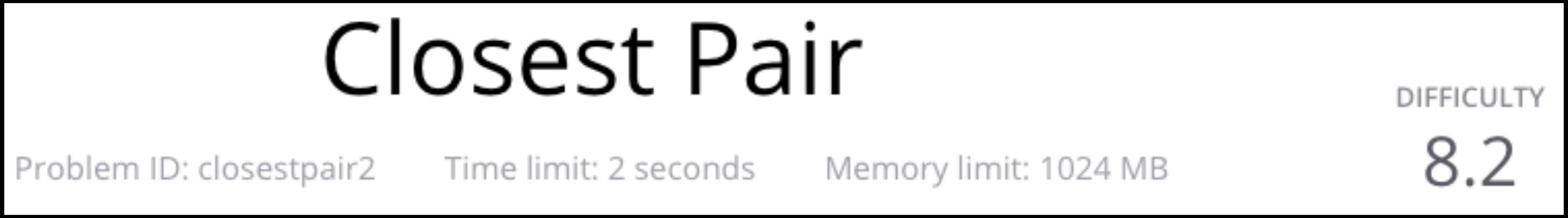
```java
// Make sure we can make it all way to the end (and add the last segment, if necessary)
if (currentPosition < l && (bestSegment == null || bestSegment.end < l))
  System.out.println(-1);
else if (currentPosition >= l)
  System.out.println(nSegmentsNeeded);
else
  System.out.println(nSegmentsNeeded + 1);
```

Solution: O(n log(n))



```java
class Segment implements Comparable<Segment> {

  double start, end;

  public Segment(double start, double end) {
    this.start = start;
    this.end = end;
  }

  @Override public int compareTo(Segment other) {
    return Double.compare(start, other.start);
  }

}
```

Problem #2: Closest Pair

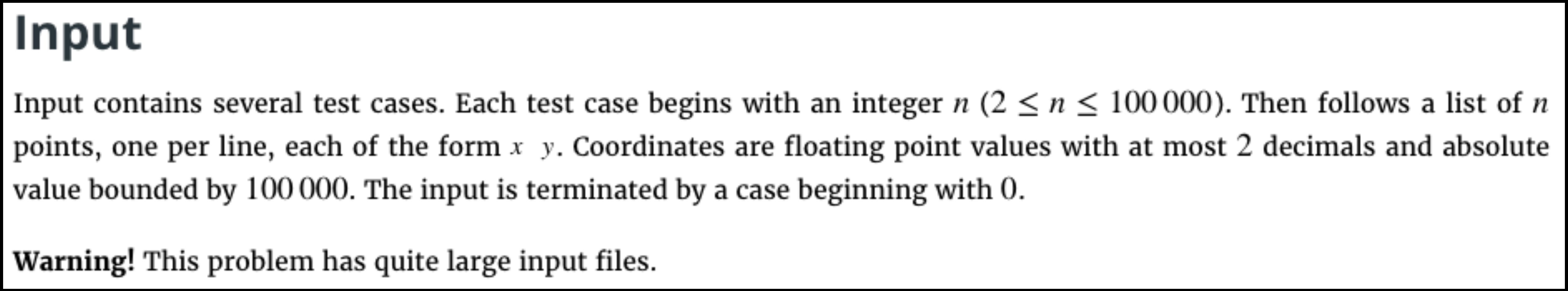


Closest Pair

Problem ID: closestpair2 Time limit: 2 seconds Memory limit: 1024 MB

DIFFICULTY
8.2

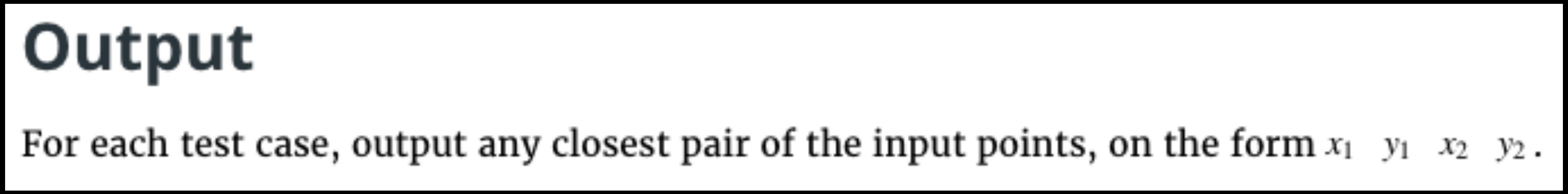
Input and Output



Input

Input contains several test cases. Each test case begins with an integer n ($2 \leq n \leq 100\,000$). Then follows a list of n points, one per line, each of the form $x \quad y$. Coordinates are floating point values with at most 2 decimals and absolute value bounded by $100\,000$. The input is terminated by a case beginning with 0.

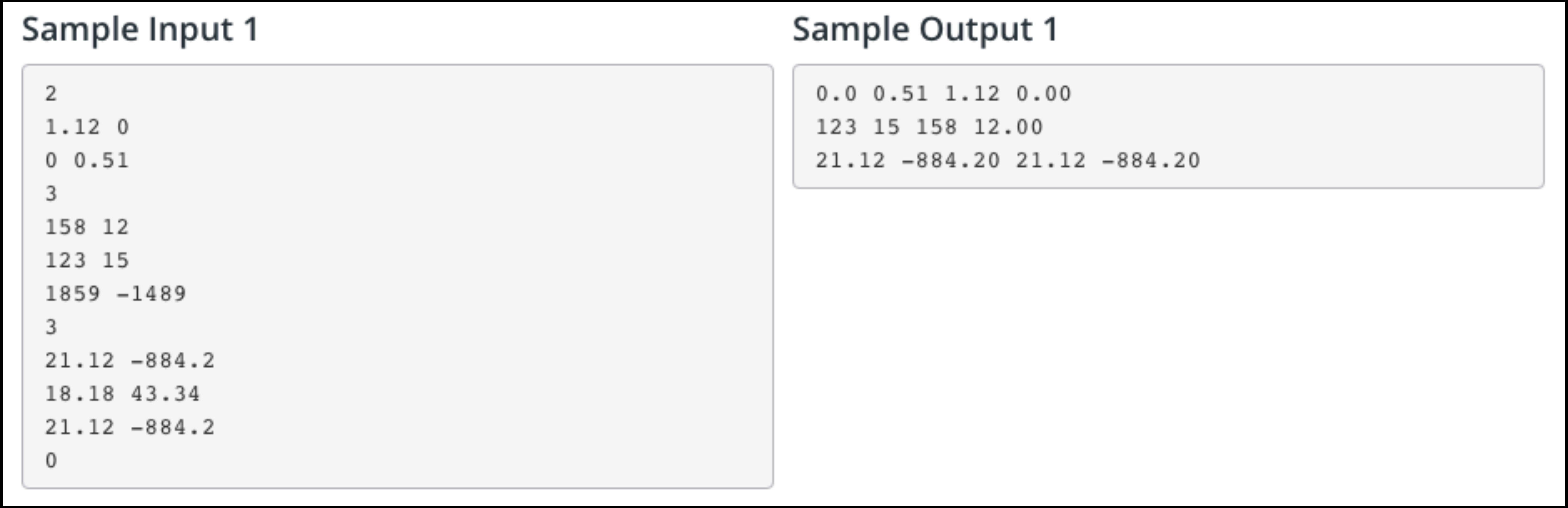
Warning! This problem has quite large input files.



Output

For each test case, output any closest pair of the input points, on the form $x_1 \quad y_1 \quad x_2 \quad y_2$.

Test Data



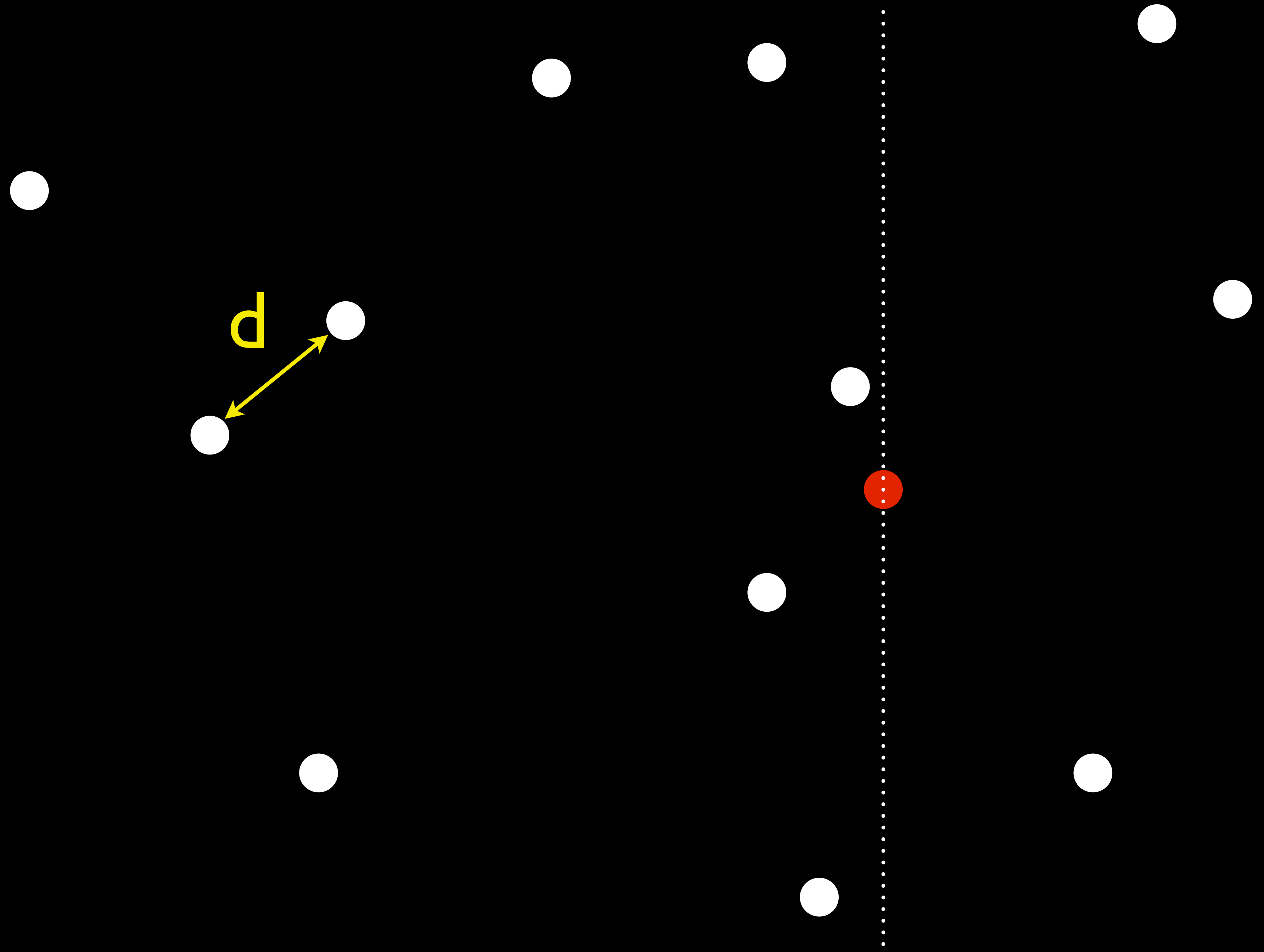
Sample Input 1

```
2
1.12 0
0 0.51
3
158 12
123 15
1859 -1489
3
21.12 -884.2
18.18 43.34
21.12 -884.2
0
```

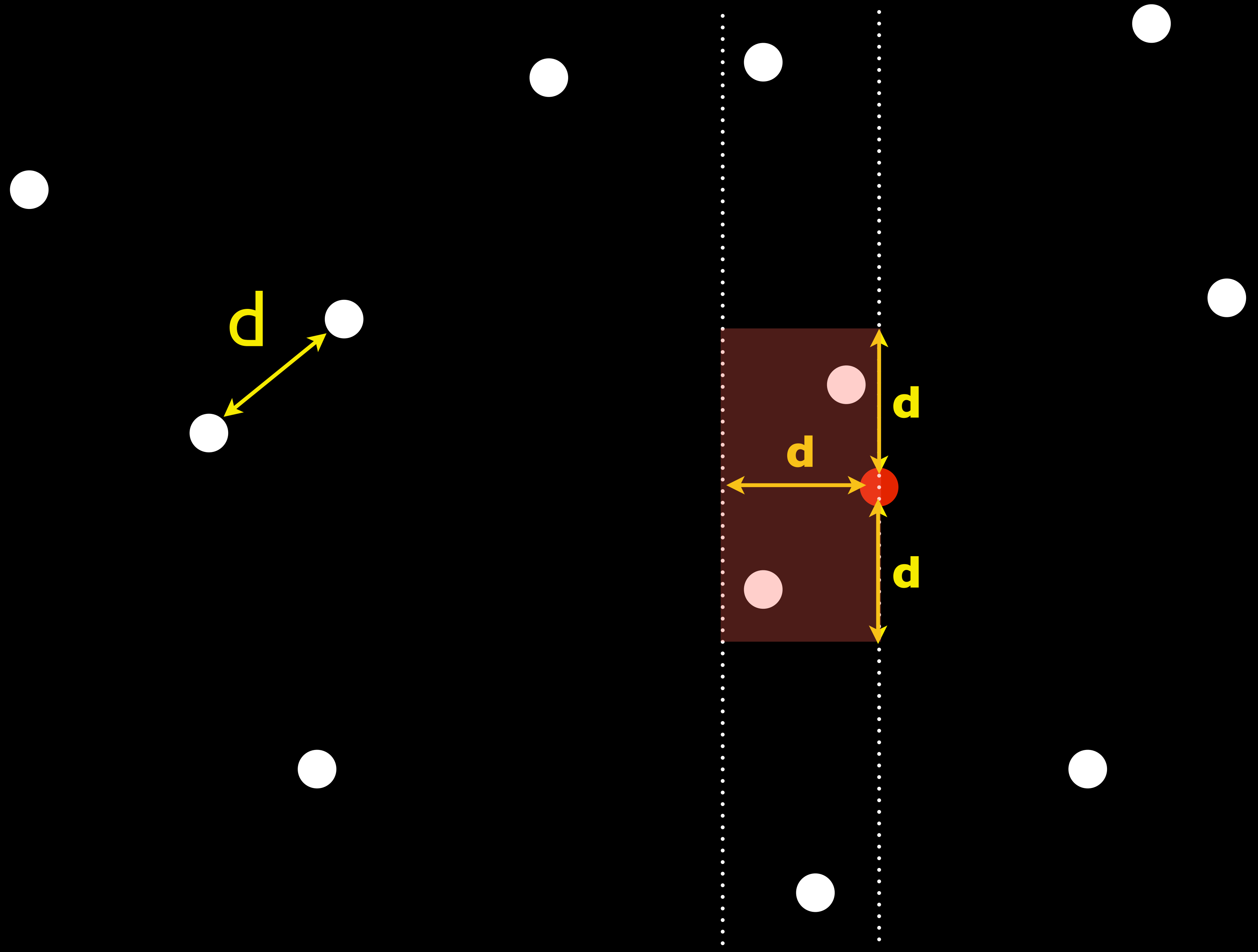
Sample Output 1

```
0.0 0.51 1.12 0.00
123 15 158 12.00
21.12 -884.20 21.12 -884.20
```

Approach



Approach



Solution: O(n log(n))

```java
// Process input
Queue<Point2D> pts = new PriorityQueue<Point2D>(n, new HorizontalComparator());
for (int i = 0; i < n; i++) {
  String[] line = br.readLine().split(" ");
  double x = Double.valueOf(line[0]);
  double y = Double.valueOf(line[1]);
  pts.add(new Point2D.Double(x, y));
}

// Sweep Left to Right
double d = Double.MAX_VALUE;
Point2D pt1 = null, pt2 = null;
Queue<Point2D> ptsInStripSortedX = new PriorityQueue<Point2D>(n, new HorizontalComparator());
TreeSet<Point2D> ptsInStripSortedY = new TreeSet<Point2D>(new VerticalComparator());
while (pts.size() > 0) {

  . . .

}

// Print the final answer
System.out.println(pt1.getX() + " " + pt1.getY() + " " + pt2.getX() + " " + pt2.getY());
```

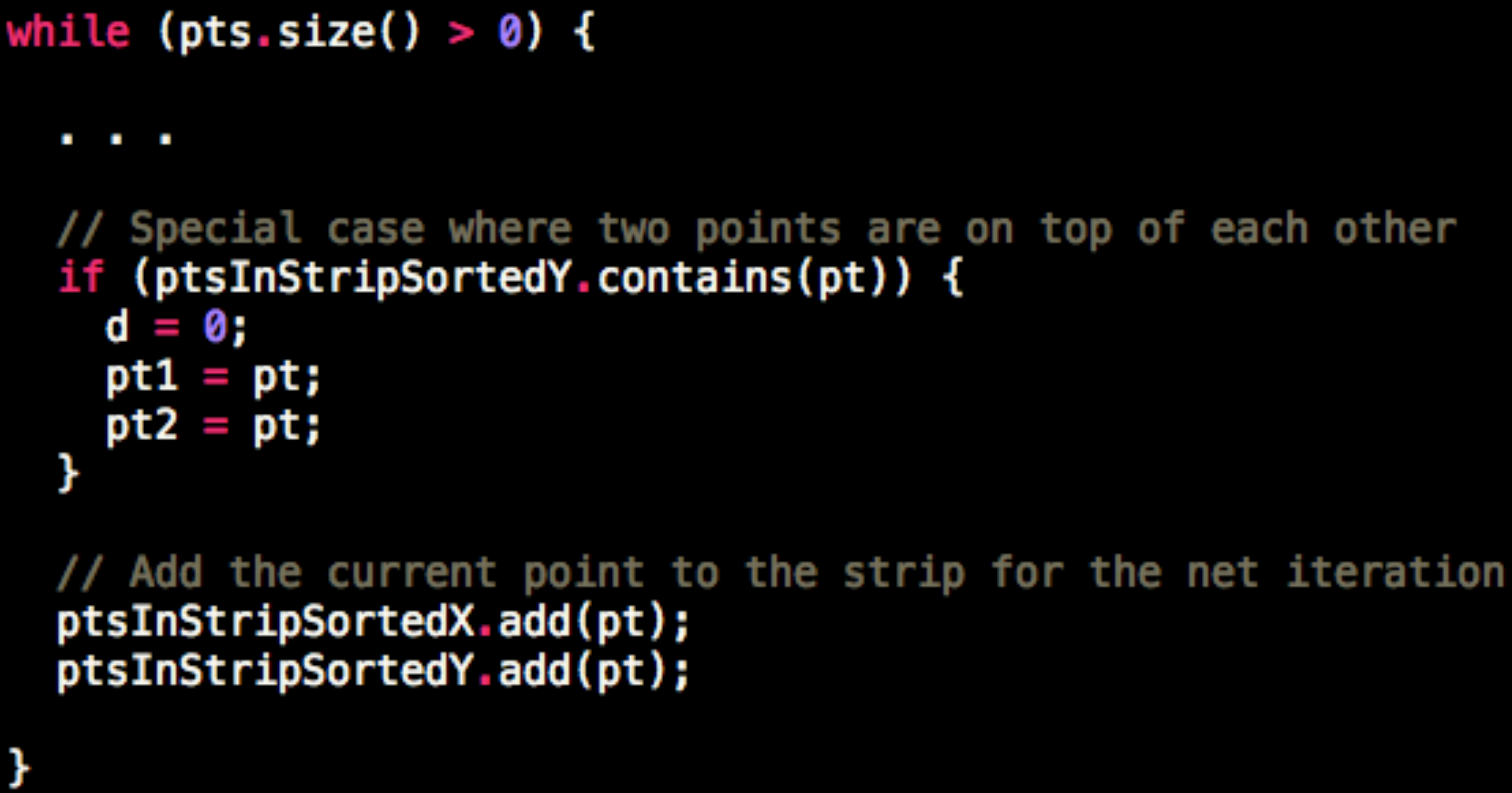
Solution: O(n log(n))

```
while (pts.size() > 0) {

  // Grab the next point to the right, which may move the strip to the right
  Point2D pt = pts.poll();

  // Remove all of the points that are no longer in the strip
  while (ptsInStripSortedX.peek() != null && ptsInStripSortedX.peek().getX() < pt.getX() - d) {
    ptsInStripSortedY.remove(ptsInStripSortedX.poll());
  }

  . . .

}
```

Solution: O(n log(n))

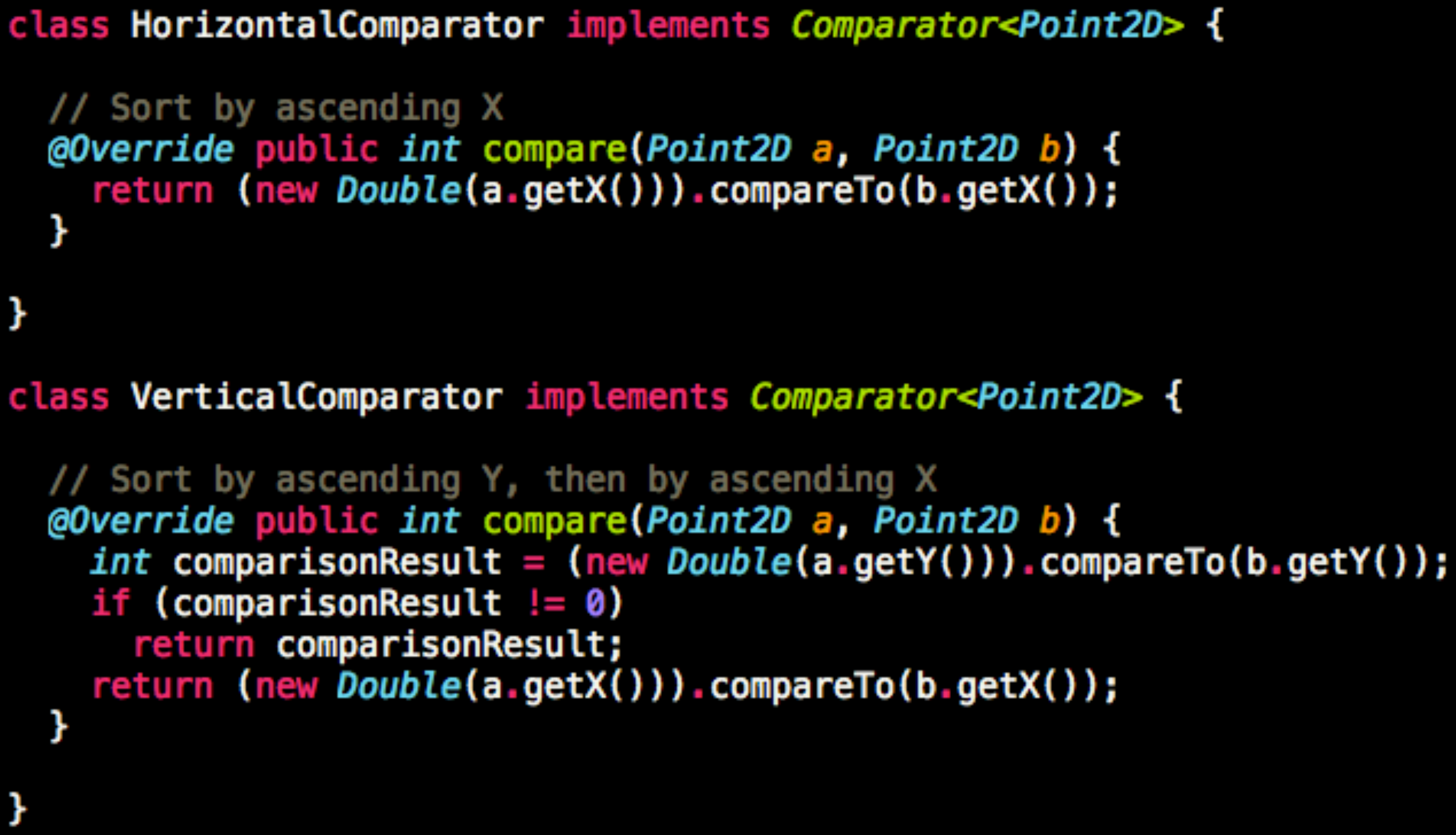
```
while (pts.size() > 0) {

  . . .

  // Check all points to the left of it that are within vertical distance 'd'
  // NOTE: The number of points in this range is O(1)
  Point2D above = ptsInStripSortedY.higher(pt);
  while (above != null && above.getY() <= pt.getY() + d) {
    double distance = pt.distance(above);
    if (distance < d) {
      d = distance;
      pt1 = pt;
      pt2 = above;
    }
    above = ptsInStripSortedY.higher(above);
  }
  Point2D below = ptsInStripSortedY.lower(pt);
  while (below != null && below.getY() >= pt.getY() - d) {
    double distance = pt.distance(below);
    if (distance < d) {
      d = distance;
      pt1 = pt;
      pt2 = below;
    }
    below = ptsInStripSortedY.higher(below);
  }

  . . .

}
```

Solution: O(n log(n))



```
while (pts.size() > 0) {

  . . .

  // Special case where two points are on top of each other
  if (ptsInStripSortedY.contains(pt)) {
    d = 0;
    pt1 = pt;
    pt2 = pt;
  }

  // Add the current point to the strip for the net iteration
  ptsInStripSortedX.add(pt);
  ptsInStripSortedY.add(pt);

}
```

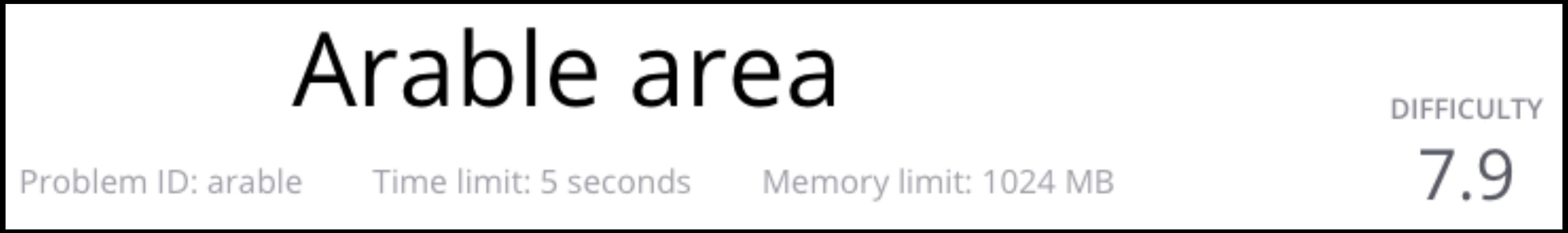
Solution: O(n log(n))



```java
class HorizontalComparator implements Comparator<Point2D> {

  // Sort by ascending X
  @Override public int compare(Point2D a, Point2D b) {
    return (new Double(a.getX())).compareTo(b.getX());
  }

}

class VerticalComparator implements Comparator<Point2D> {

  // Sort by ascending Y, then by ascending X
  @Override public int compare(Point2D a, Point2D b) {
    int comparisonResult = (new Double(a.getY())).compareTo(b.getY());
    if (comparisonResult != 0)
      return comparisonResult;
    return (new Double(a.getX())).compareTo(b.getX());
  }

}
```

]

Problem #3: Arable Area

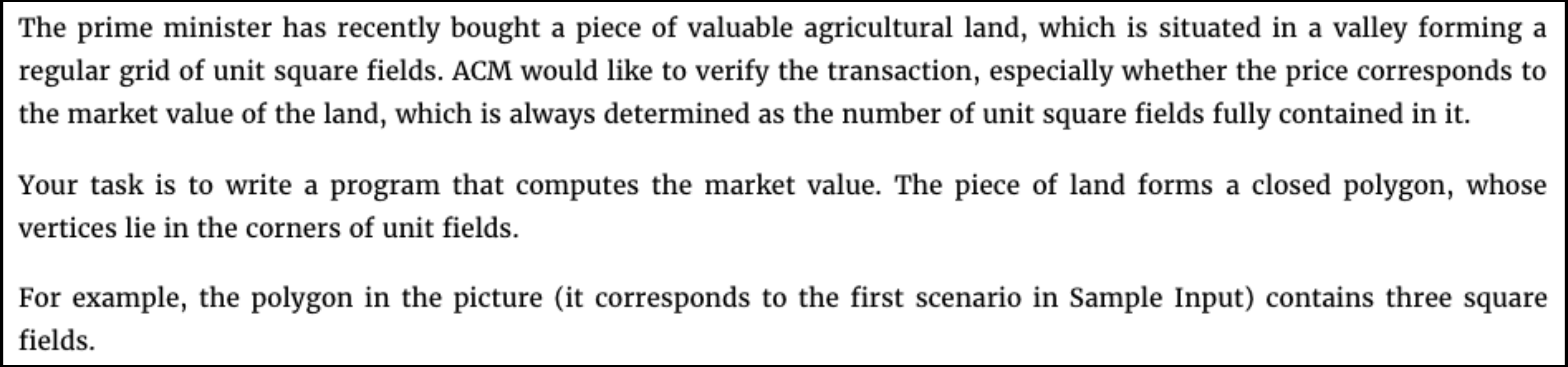


Arable area

Problem ID: arable Time limit: 5 seconds Memory limit: 1024 MB

DIFFICULTY
7.9

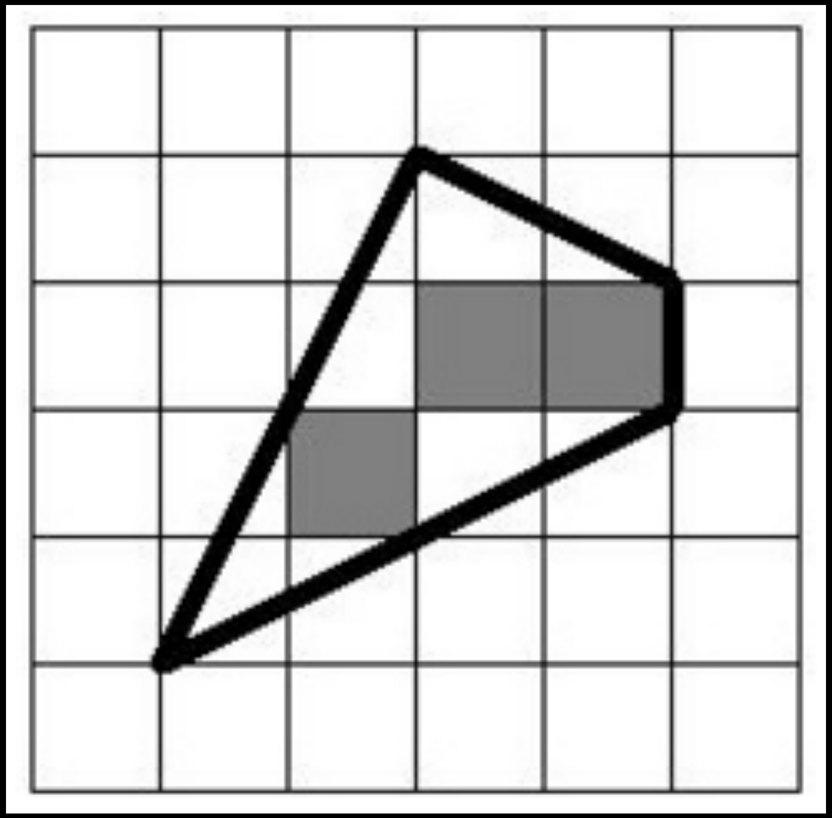
Problem Description



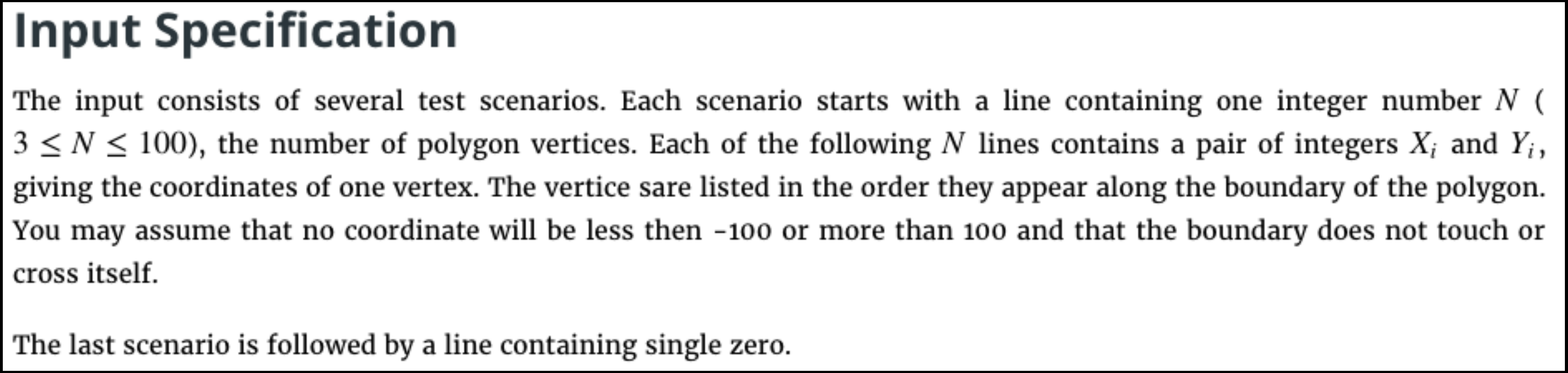
The prime minister has recently bought a piece of valuable agricultural land, which is situated in a valley forming a regular grid of unit square fields. ACM would like to verify the transaction, especially whether the price corresponds to the market value of the land, which is always determined as the number of unit square fields fully contained in it.

Your task is to write a program that computes the market value. The piece of land forms a closed polygon, whose vertices lie in the corners of unit fields.

For example, the polygon in the picture (it corresponds to the first scenario in Sample Input) contains three square fields.



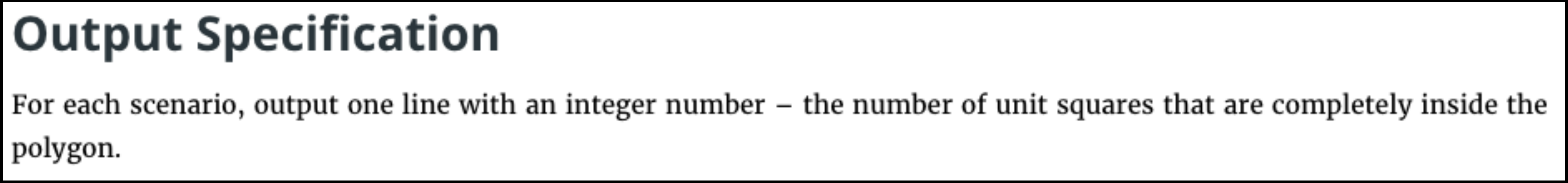
Input and Output



Input Specification

The input consists of several test scenarios. Each scenario starts with a line containing one integer number N ($3 \leq N \leq 100$), the number of polygon vertices. Each of the following N lines contains a pair of integers X_i and Y_i, giving the coordinates of one vertex. The vertice sare listed in the order they appear along the boundary of the polygon. You may assume that no coordinate will be less then -100 or more than 100 and that the boundary does not touch or cross itself.

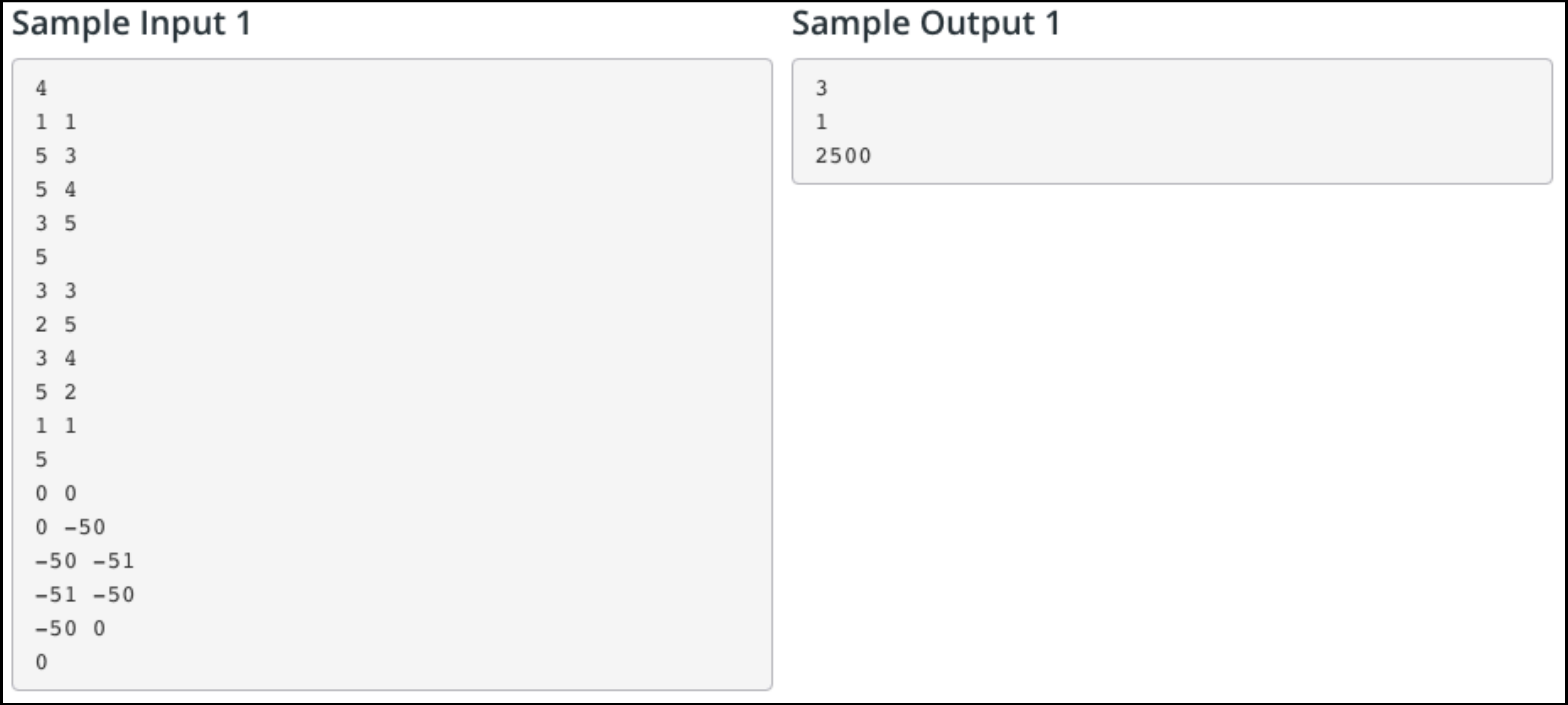
The last scenario is followed by a line containing single zero.



Output Specification

For each scenario, output one line with an integer number – the number of unit squares that are completely inside the polygon.

Test Data



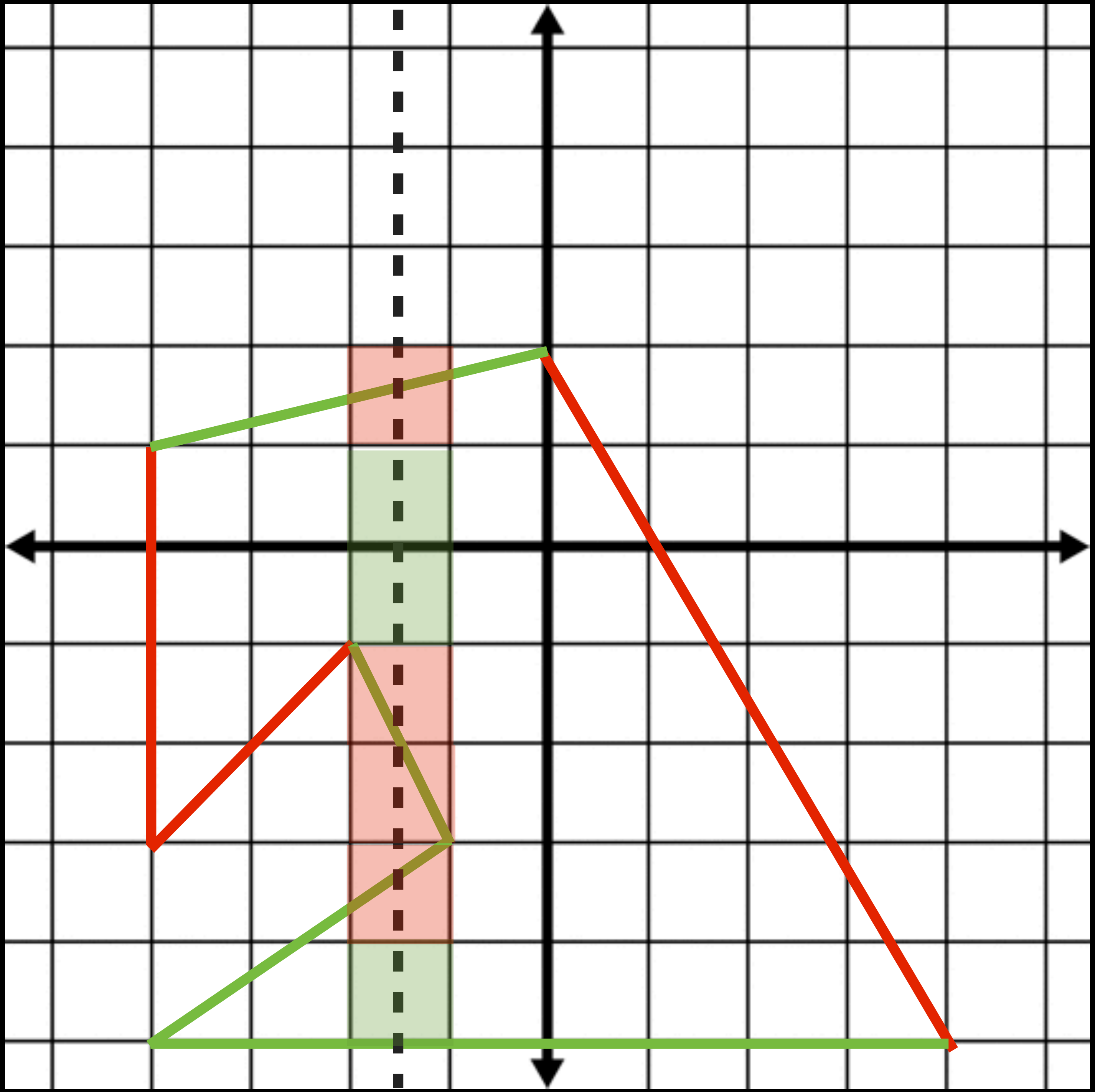
Sample Input 1

```
4
1 1
5 3
5 4
3 5
5
3 3
2 5
3 4
5 2
1 1
5
0 0
0 -50
-50 -51
-51 -50
-50 0
0
```

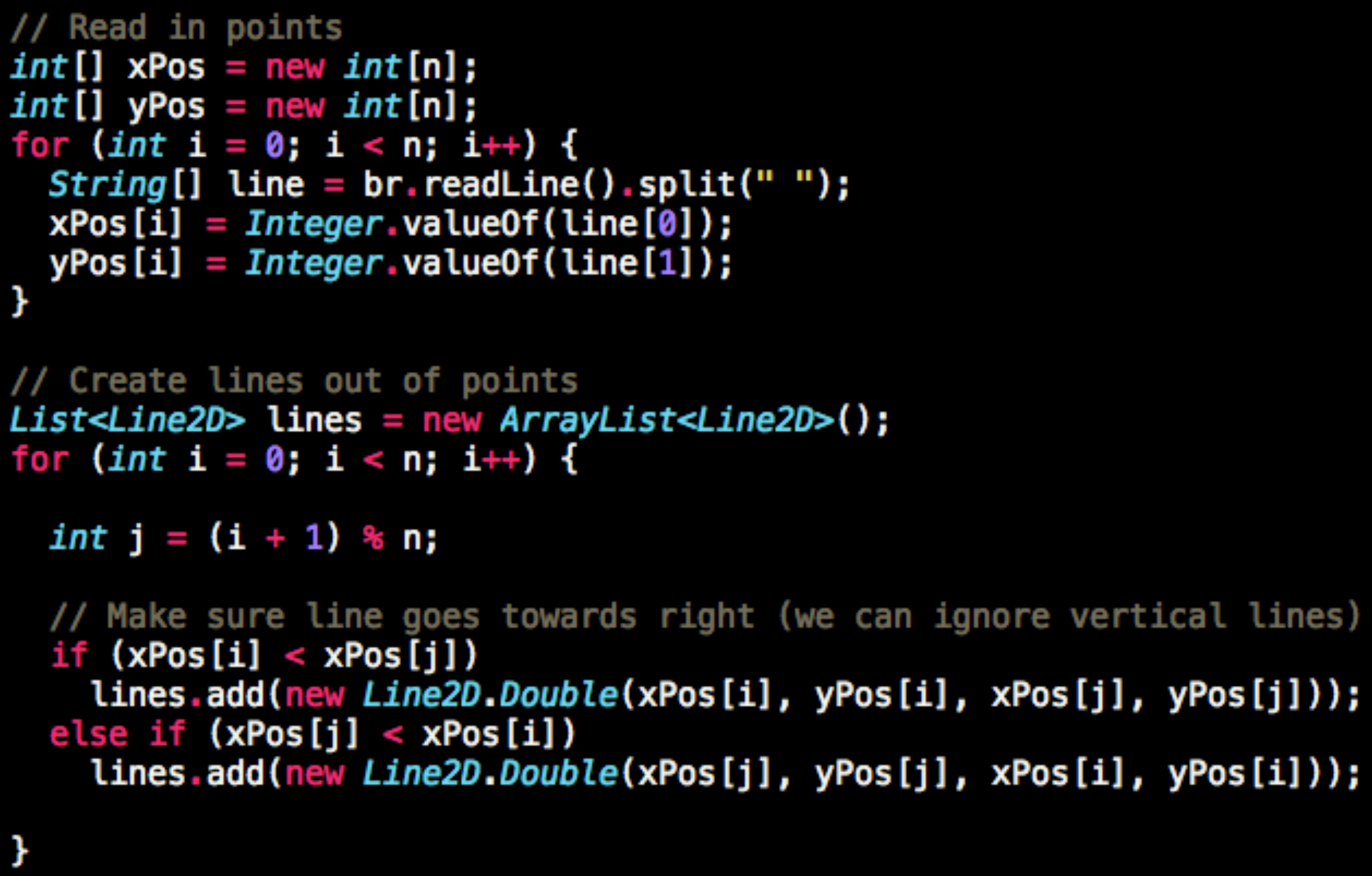
Sample Output 1

```
3
1
2500
```

Approach

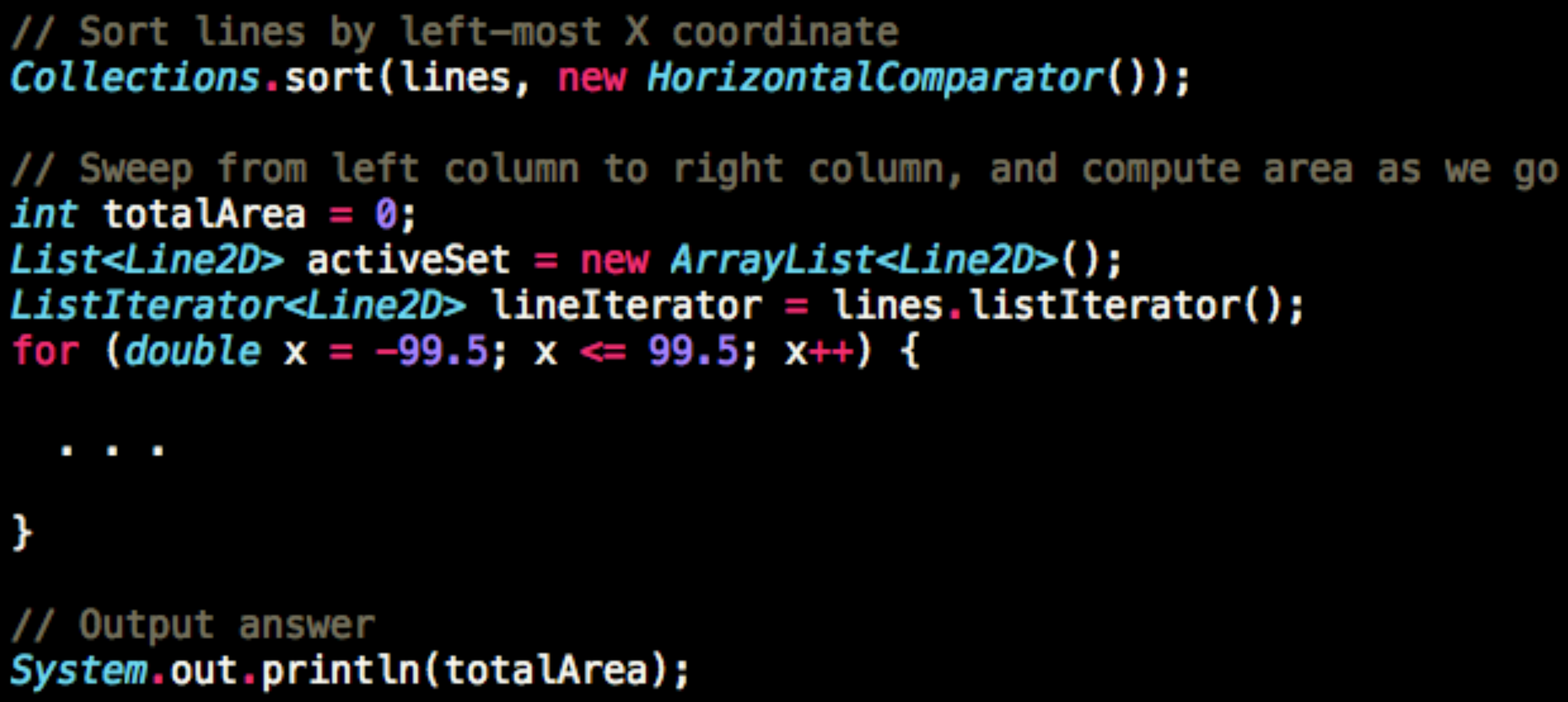


Solution: O(w n log(n))



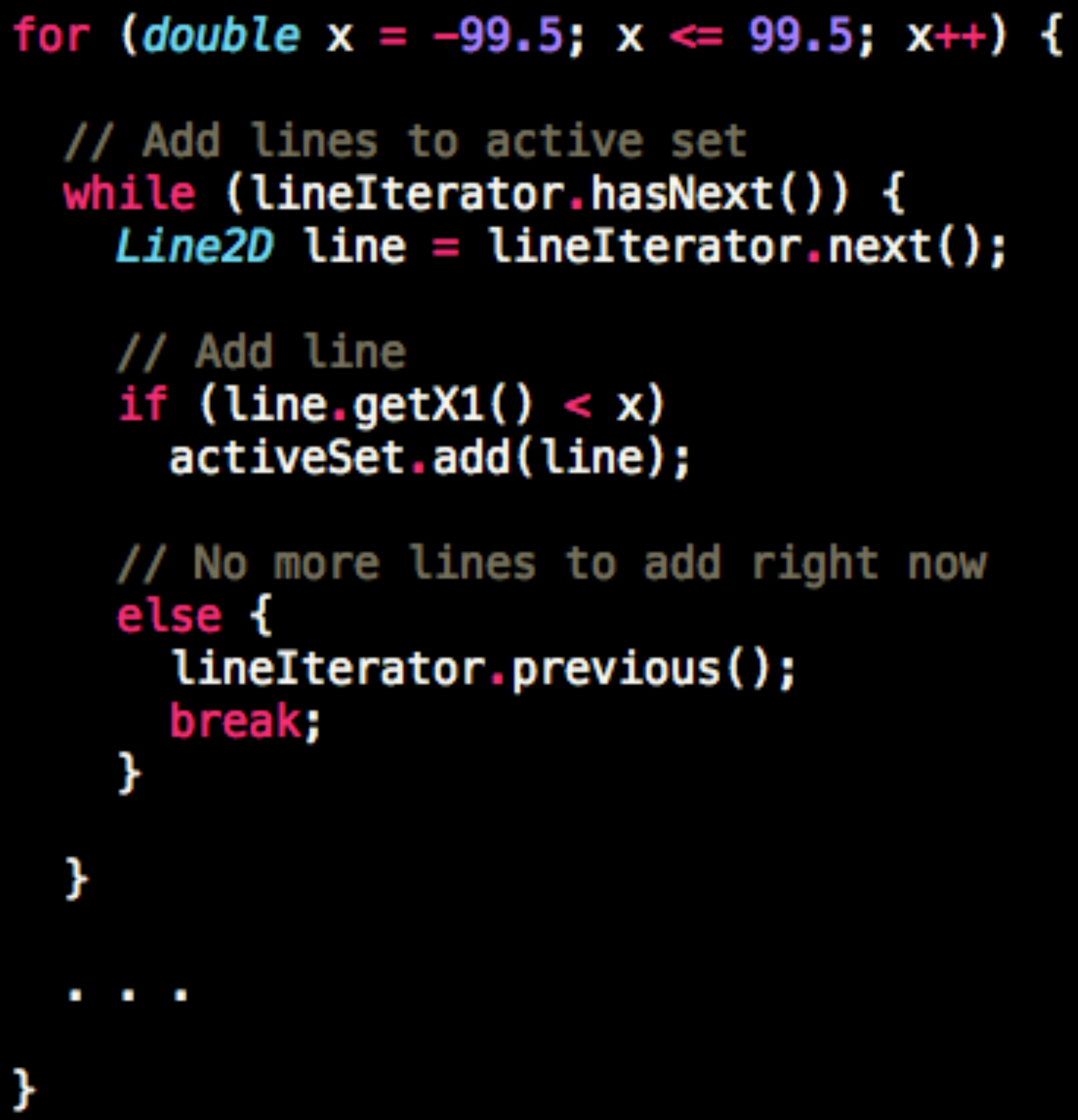
```java
// Read in points
int[] xPos = new int[n];
int[] yPos = new int[n];
for (int i = 0; i < n; i++) {
  String[] line = br.readLine().split(" ");
  xPos[i] = Integer.valueOf(line[0]);
  yPos[i] = Integer.valueOf(line[1]);
}

// Create lines out of points
List<Line2D> lines = new ArrayList<Line2D>();
for (int i = 0; i < n; i++) {

  int j = (i + 1) % n;

  // Make sure line goes towards right (we can ignore vertical lines)
  if (xPos[i] < xPos[j])
    lines.add(new Line2D.Double(xPos[i], yPos[i], xPos[j], yPos[j]));
  else if (xPos[j] < xPos[i])
    lines.add(new Line2D.Double(xPos[j], yPos[j], xPos[i], yPos[i]));

}
```

Solution: O(w n log(n))



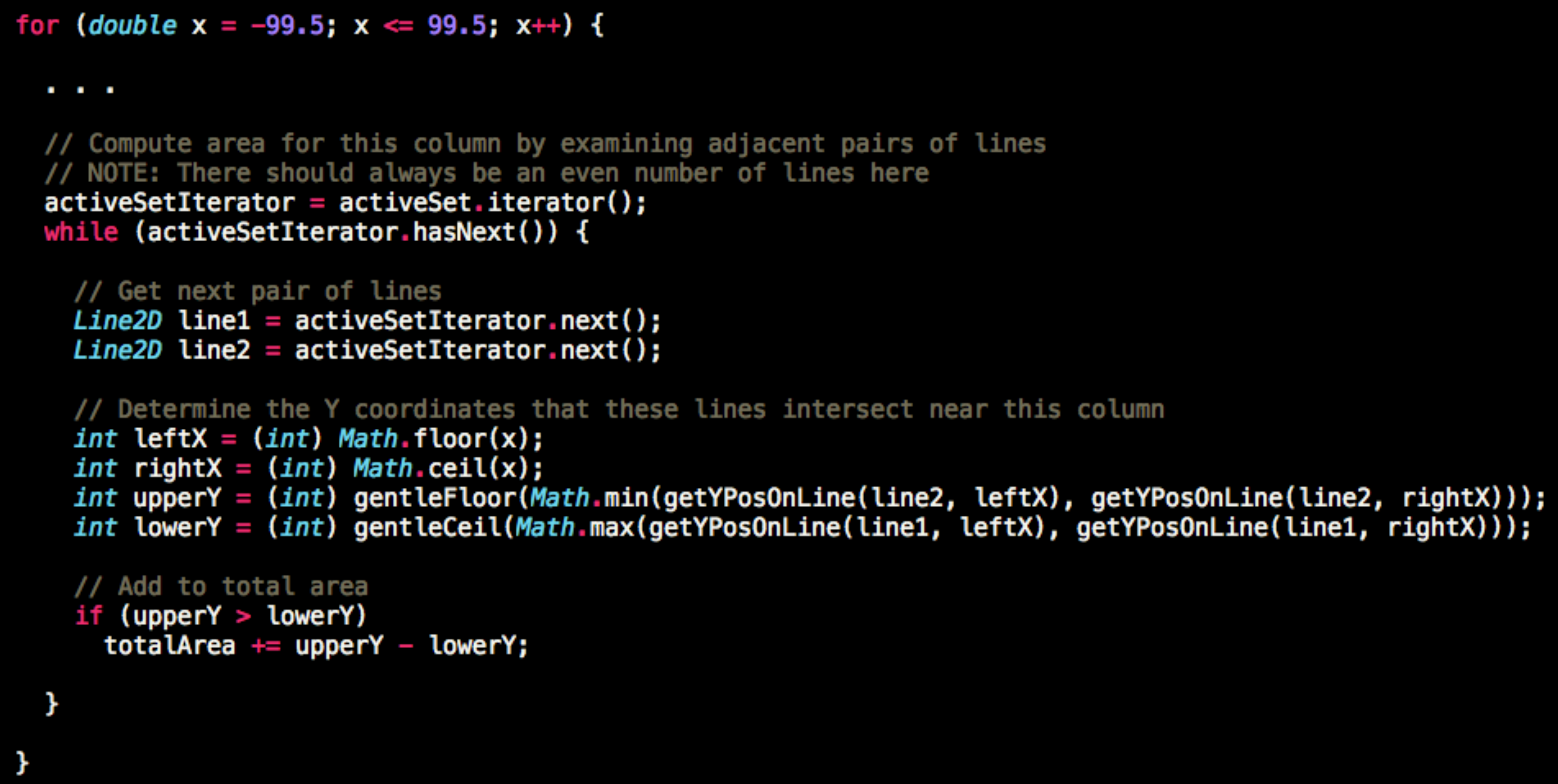
```java
// Sort lines by left-most X coordinate
Collections.sort(lines, new HorizontalComparator());

// Sweep from left column to right column, and compute area as we go
int totalArea = 0;
List<Line2D> activeSet = new ArrayList<Line2D>();
ListIterator<Line2D> lineIterator = lines.listIterator();
for (double x = -99.5; x <= 99.5; x++) {

  . . .

}

// Output answer
System.out.println(totalArea);
```

Solution: O(w n log(n))



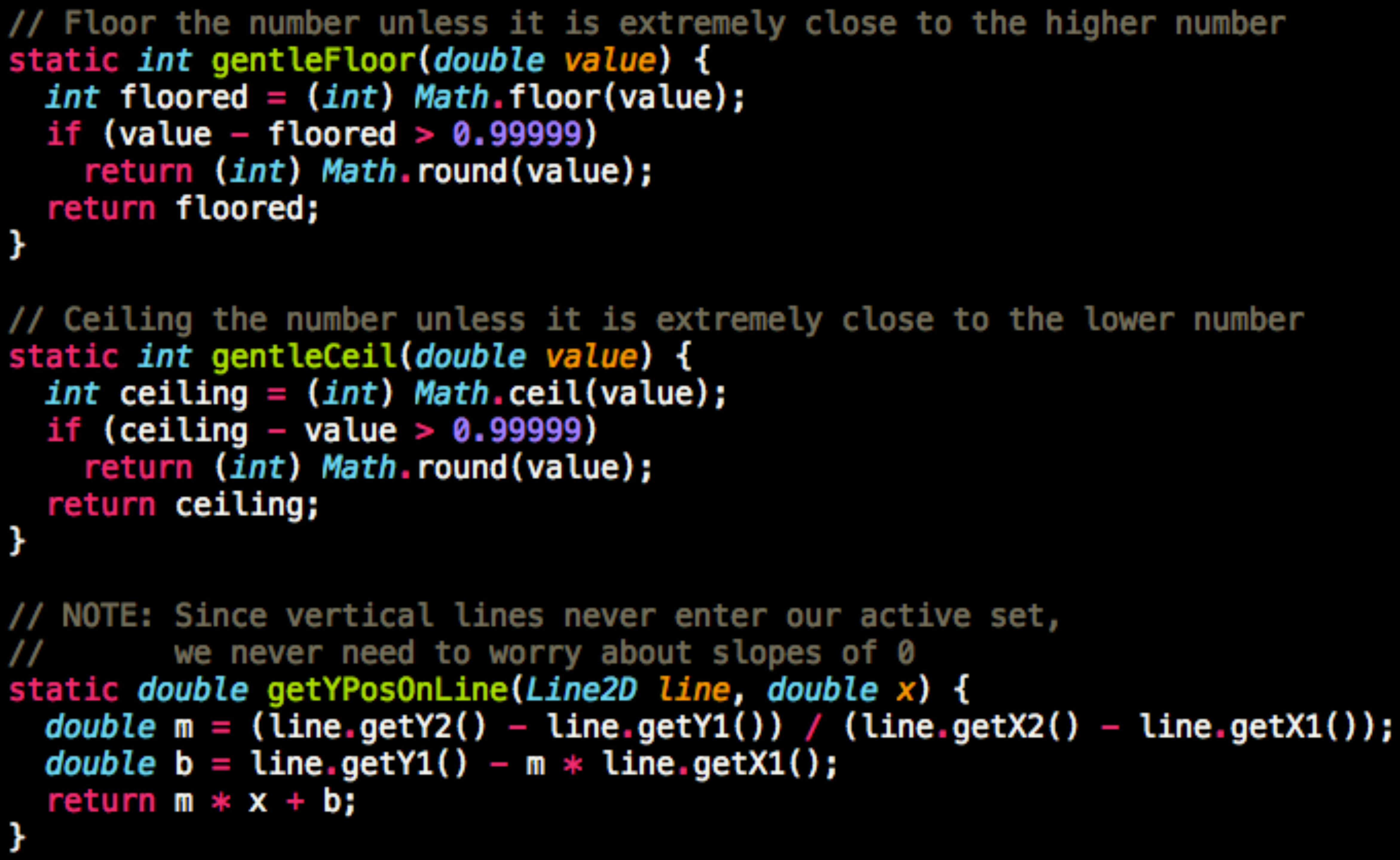
```
for (double x = -99.5; x <= 99.5; x++) {

  // Add lines to active set
  while (lineIterator.hasNext()) {
    Line2D line = lineIterator.next();

    // Add line
    if (line.getX1() < x)
      activeSet.add(line);

    // No more lines to add right now
    else {
      lineIterator.previous();
      break;
    }

  }

  . . .

}
```

Solution: O(w n log(n))



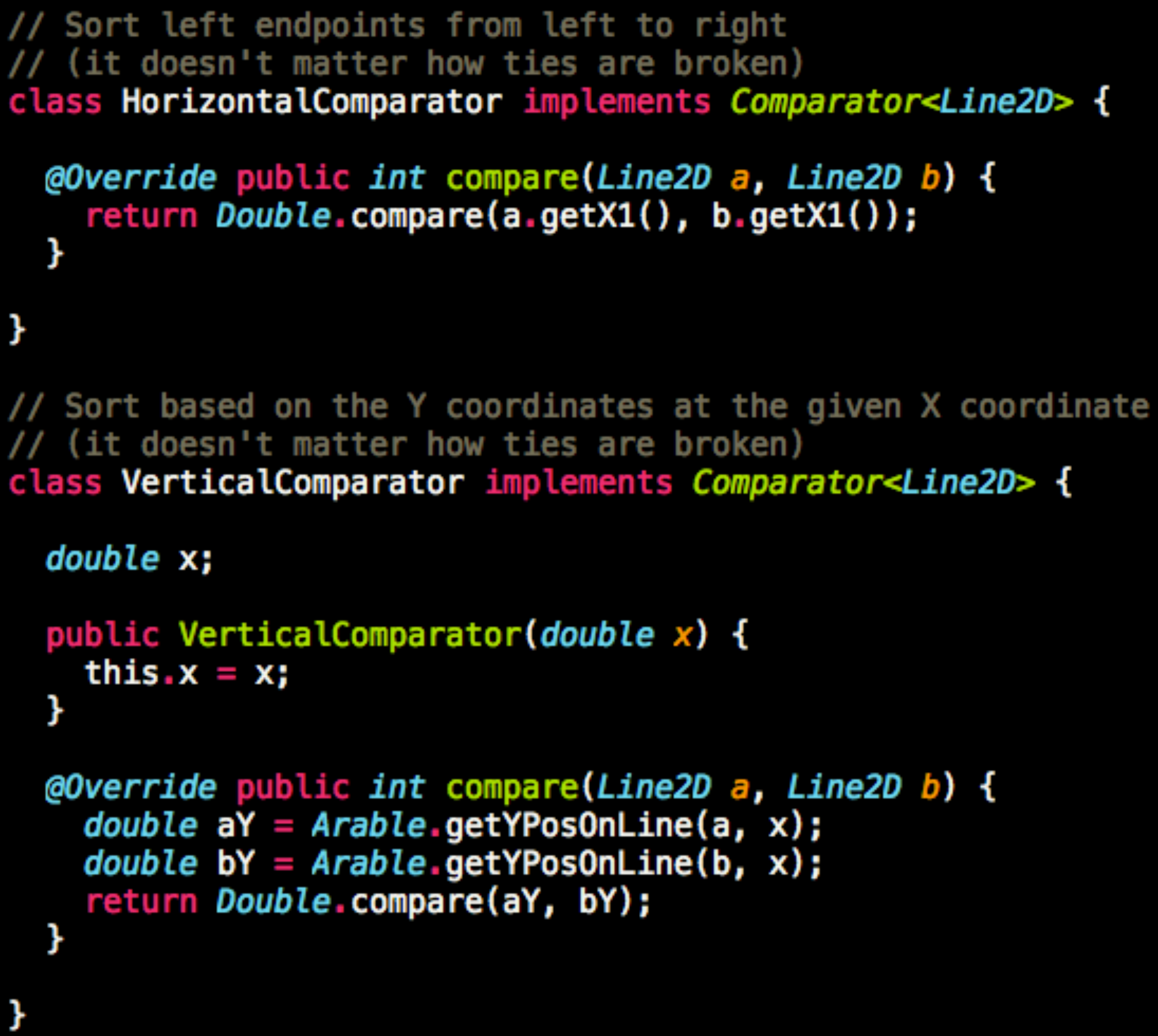
```
for (double x = -99.5; x <= 99.5; x++) {

  . . .

  // Compute area for this column by examining adjacent pairs of lines
  // NOTE: There should always be an even number of lines here
  activeSetIterator = activeSet.iterator();
  while (activeSetIterator.hasNext()) {

    // Get next pair of lines
    Line2D line1 = activeSetIterator.next();
    Line2D line2 = activeSetIterator.next();

    // Determine the Y coordinates that these lines intersect near this column
    int leftX = (int) Math.floor(x);
    int rightX = (int) Math.ceil(x);
    int upperY = (int) gentleFloor(Math.min(getYPosOnLine(line2, leftX), getYPosOnLine(line2, rightX)));
    int lowerY = (int) gentleCeil(Math.max(getYPosOnLine(line1, leftX), getYPosOnLine(line1, rightX)));

    // Add to total area
    if (upperY > lowerY)
      totalArea += upperY - lowerY;

  }

}
```

Solution: O(w n log(n))



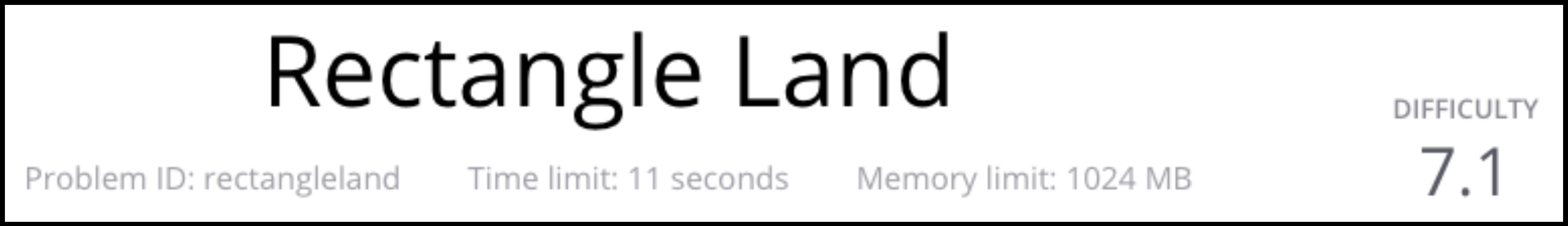
```java
// Floor the number unless it is extremely close to the higher number
static int gentleFloor(double value) {
  int floored = (int) Math.floor(value);
  if (value - floored > 0.99999)
    return (int) Math.round(value);
  return floored;
}

// Ceiling the number unless it is extremely close to the lower number
static int gentleCeil(double value) {
  int ceiling = (int) Math.ceil(value);
  if (ceiling - value > 0.99999)
    return (int) Math.round(value);
  return ceiling;
}

// NOTE: Since vertical lines never enter our active set,
//       we never need to worry about slopes of 0
static double getYPosOnLine(Line2D line, double x) {
  double m = (line.getY2() - line.getY1()) / (line.getX2() - line.getX1());
  double b = line.getY1() - m * line.getX1();
  return m * x + b;
}
```

Solution: O(w n log(n))



```java
// Sort left endpoints from left to right
// (it doesn't matter how ties are broken)
class HorizontalComparator implements Comparator<Line2D> {

  @Override public int compare(Line2D a, Line2D b) {
    return Double.compare(a.getX1(), b.getX1());
  }

}

// Sort based on the Y coordinates at the given X coordinate
// (it doesn't matter how ties are broken)
class VerticalComparator implements Comparator<Line2D> {

  double x;

  public VerticalComparator(double x) {
    this.x = x;
  }

  @Override public int compare(Line2D a, Line2D b) {
    double aY = Arable.getYPosOnLine(a, x);
    double bY = Arable.getYPosOnLine(b, x);
    return Double.compare(aY, bY);
  }

}
```

Problem #4: Rectangle Land

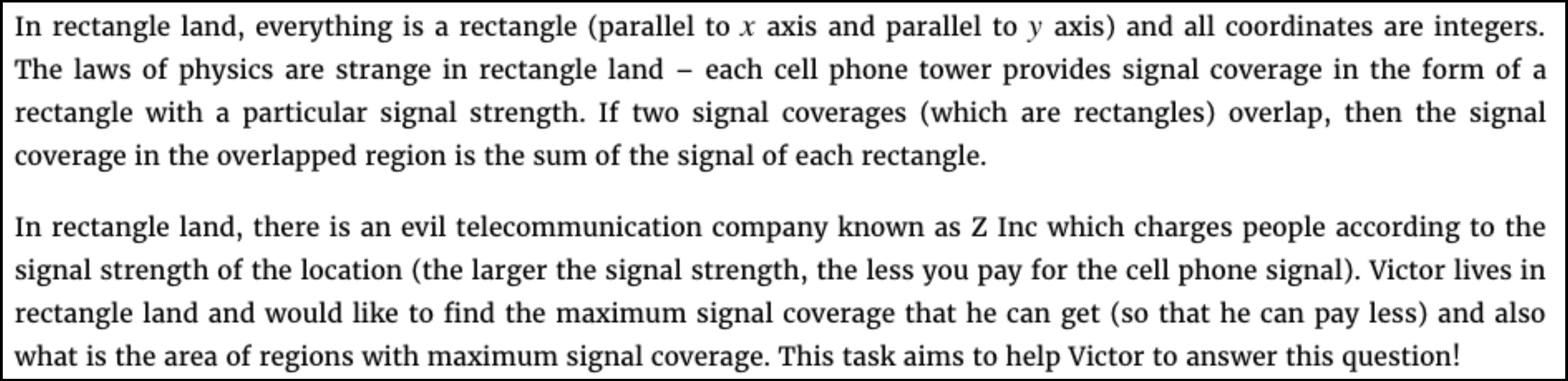


Rectangle Land

Problem ID: rectangleland Time limit: 11 seconds Memory limit: 1024 MB

DIFFICULTY 7.1

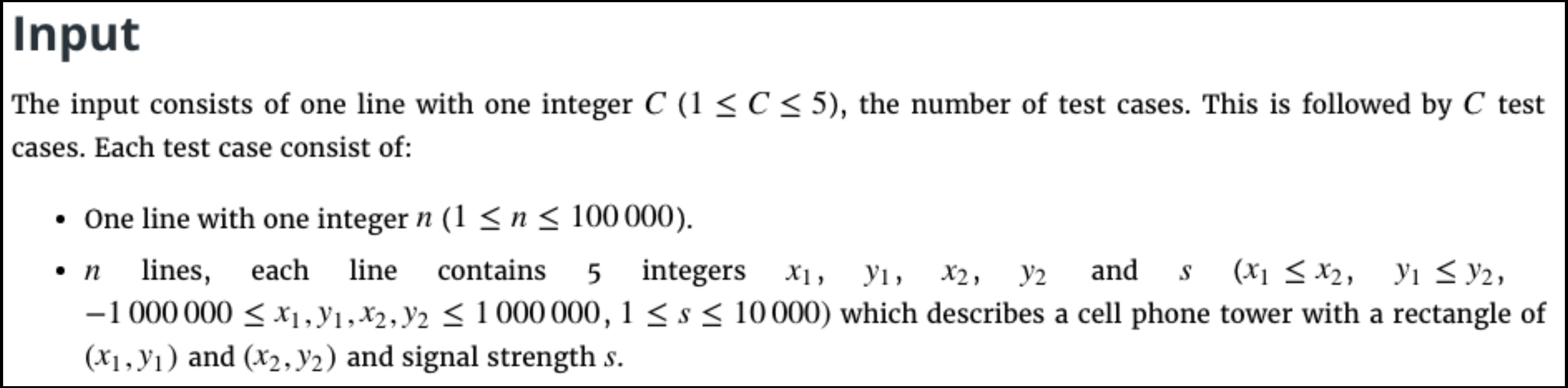
Problem Description



In rectangle land, everything is a rectangle (parallel to x axis and parallel to y axis) and all coordinates are integers. The laws of physics are strange in rectangle land – each cell phone tower provides signal coverage in the form of a rectangle with a particular signal strength. If two signal coverages (which are rectangles) overlap, then the signal coverage in the overlapped region is the sum of the signal of each rectangle.

In rectangle land, there is an evil telecommunication company known as Z Inc which charges people according to the signal strength of the location (the larger the signal strength, the less you pay for the cell phone signal). Victor lives in rectangle land and would like to find the maximum signal coverage that he can get (so that he can pay less) and also what is the area of regions with maximum signal coverage. This task aims to help Victor to answer this question!

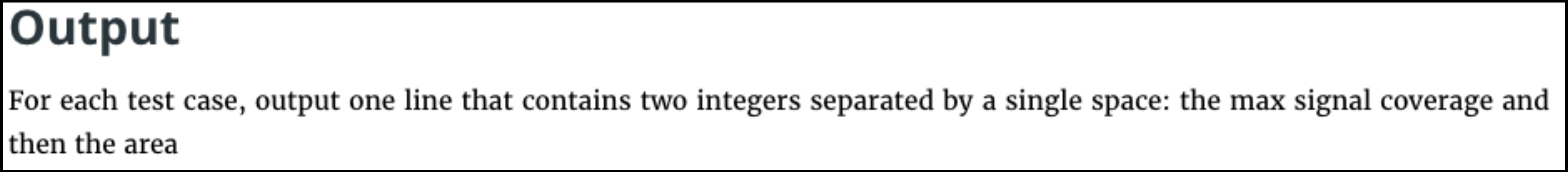
Input and Output



Input

The input consists of one line with one integer C ($1 \leq C \leq 5$), the number of test cases. This is followed by C test cases. Each test case consist of:

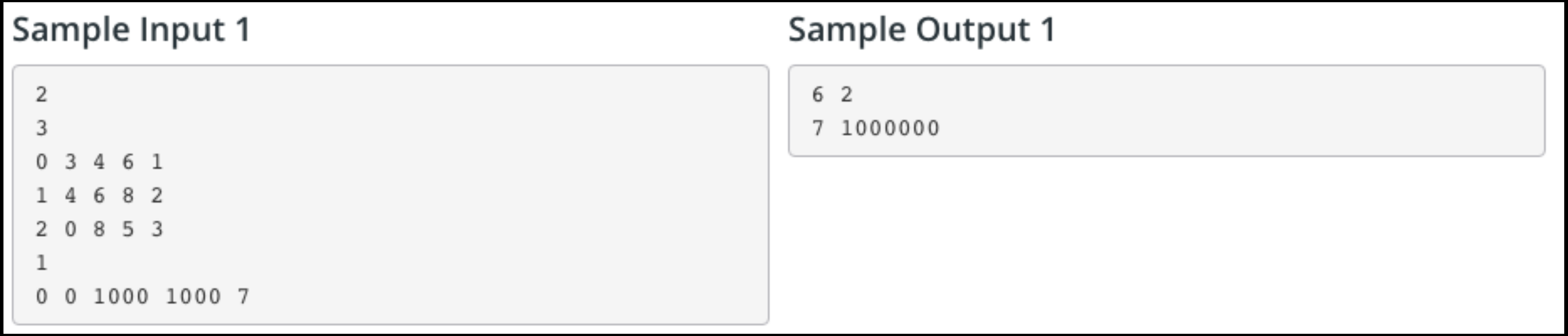
- One line with one integer n ($1 \leq n \leq 100\,000$).
- n lines, each line contains 5 integers x_1, y_1, x_2, y_2 and s ($x_1 \leq x_2$, $y_1 \leq y_2$, $-1\,000\,000 \leq x_1, y_1, x_2, y_2 \leq 1\,000\,000$, $1 \leq s \leq 10\,000$) which describes a cell phone tower with a rectangle of (x_1, y_1) and (x_2, y_2) and signal strength s.



Output

For each test case, output one line that contains two integers separated by a single space: the max signal coverage and then the area

Test Data



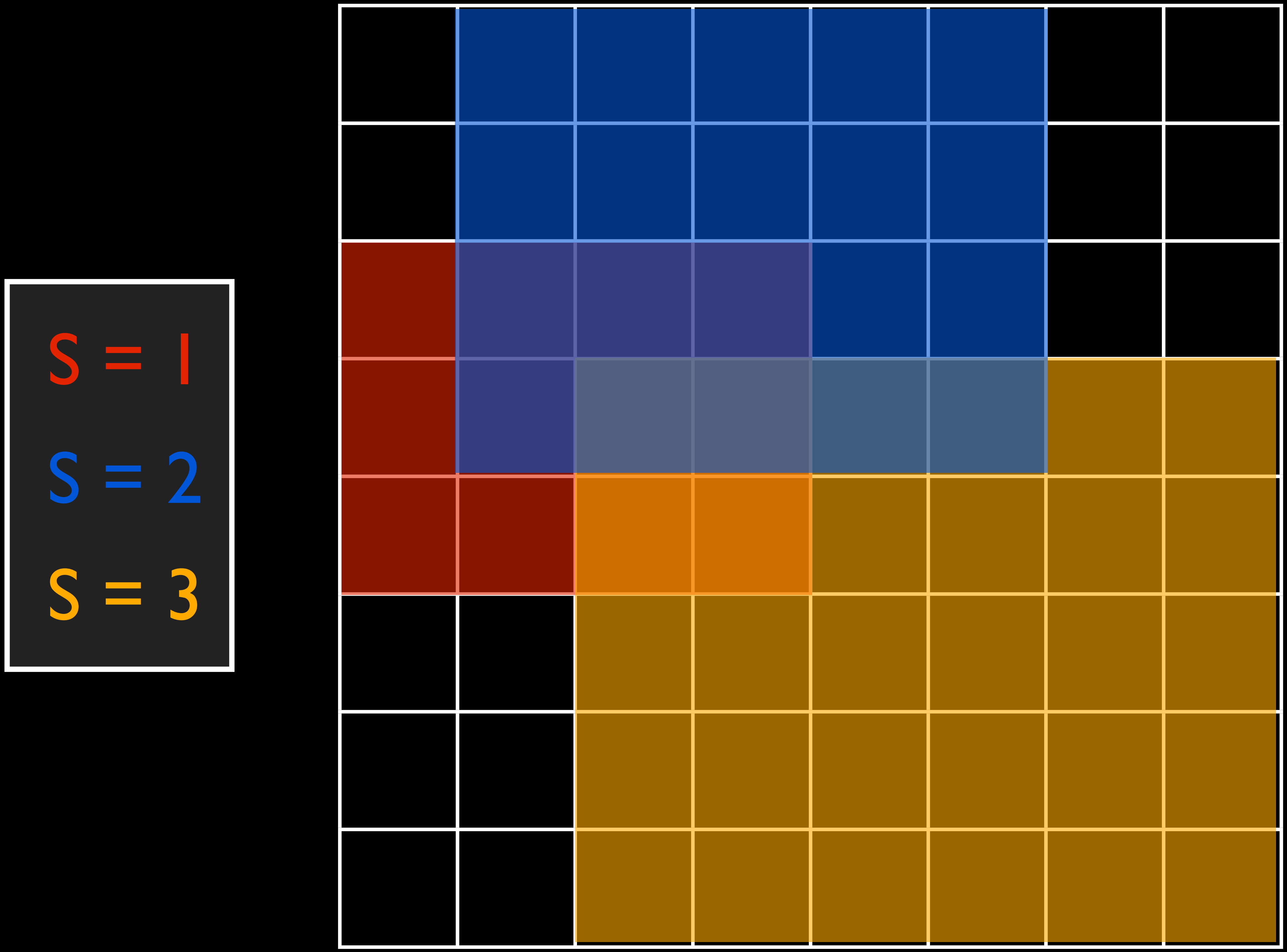
Sample Input 1

```
2
3
0 3 4 6 1
1 4 6 8 2
2 0 8 5 3
1
0 0 1000 1000 7
```

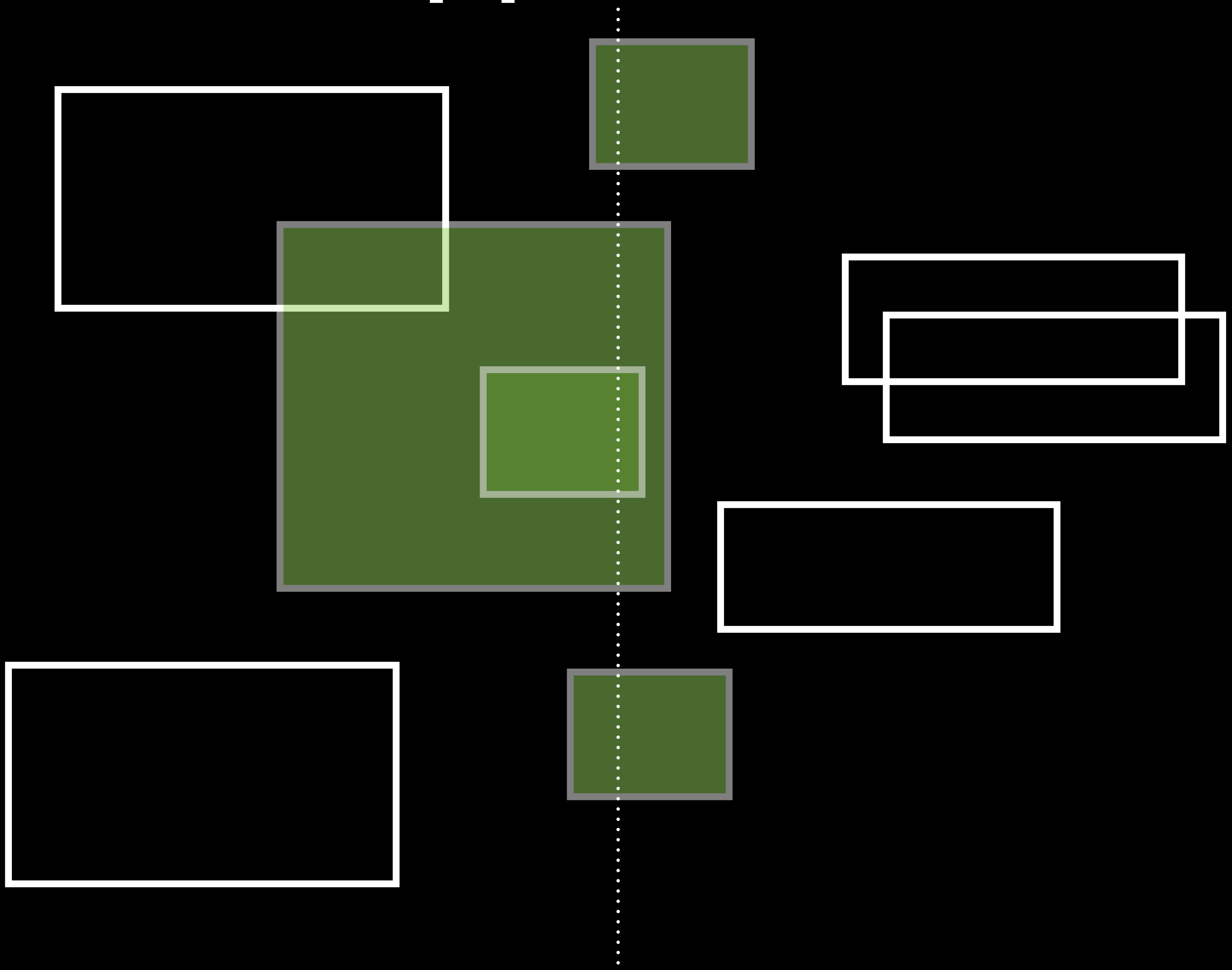
Sample Output 1

```
6 2
7 1000000
```

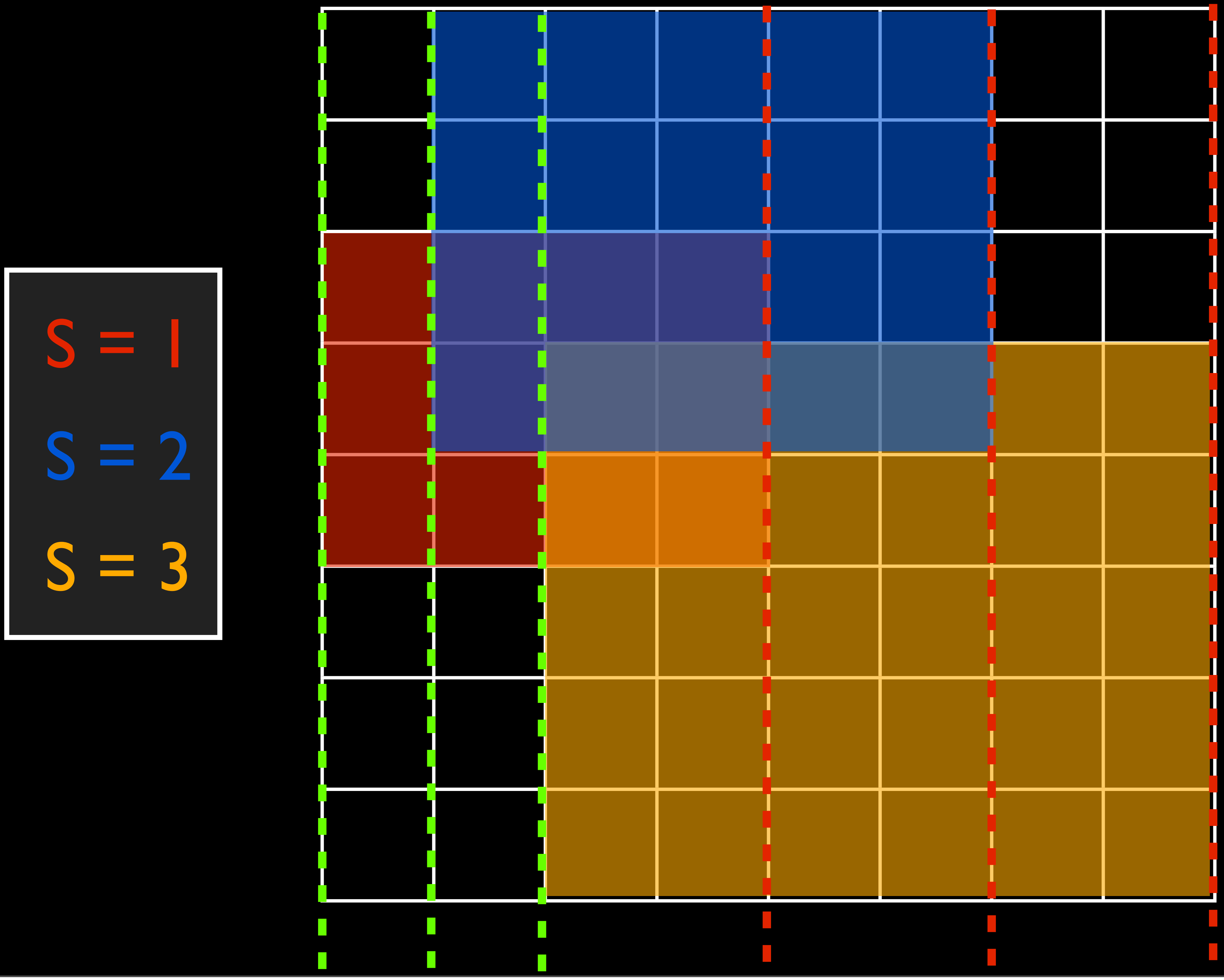
Approach



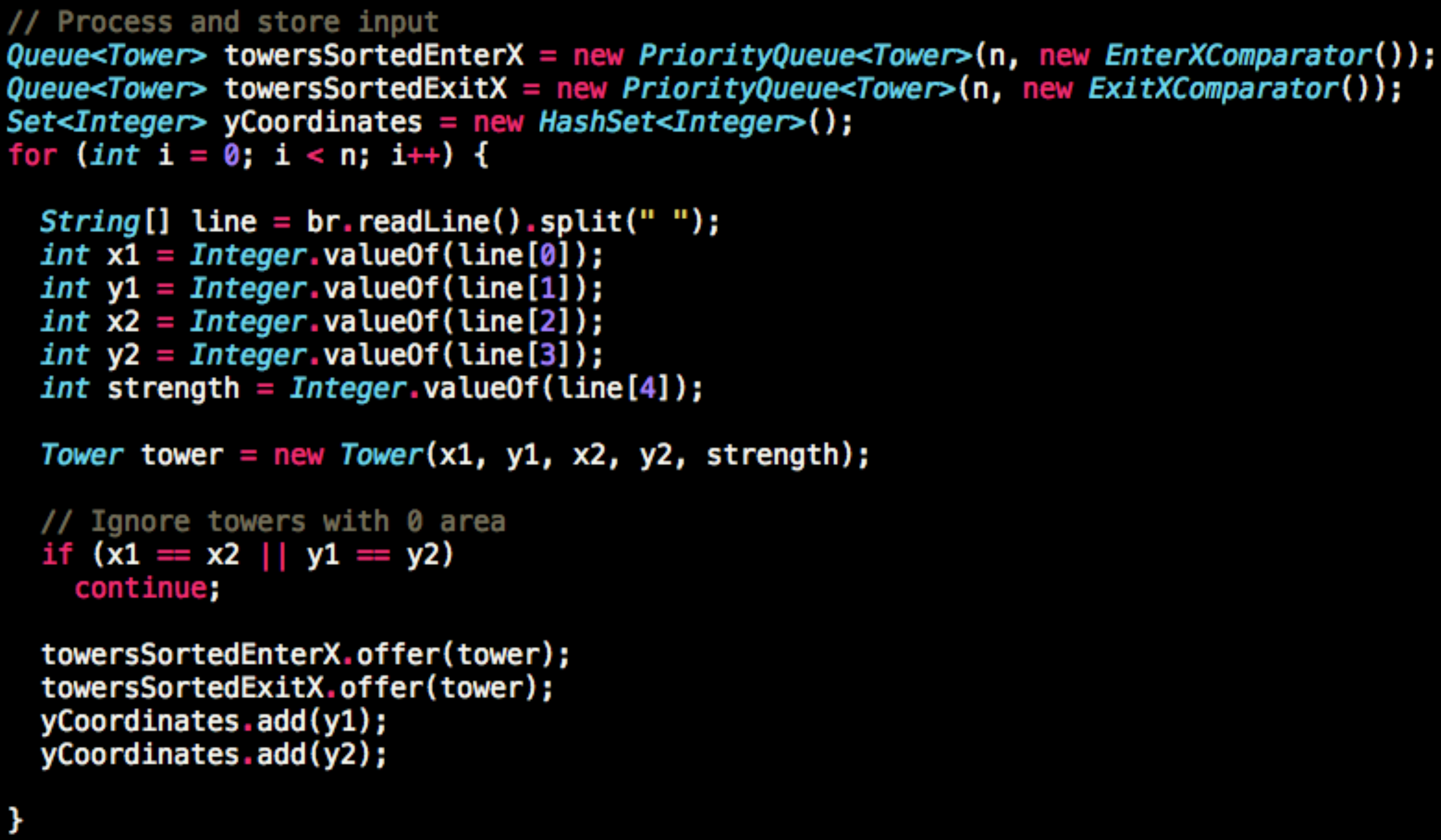
Approach



Approach

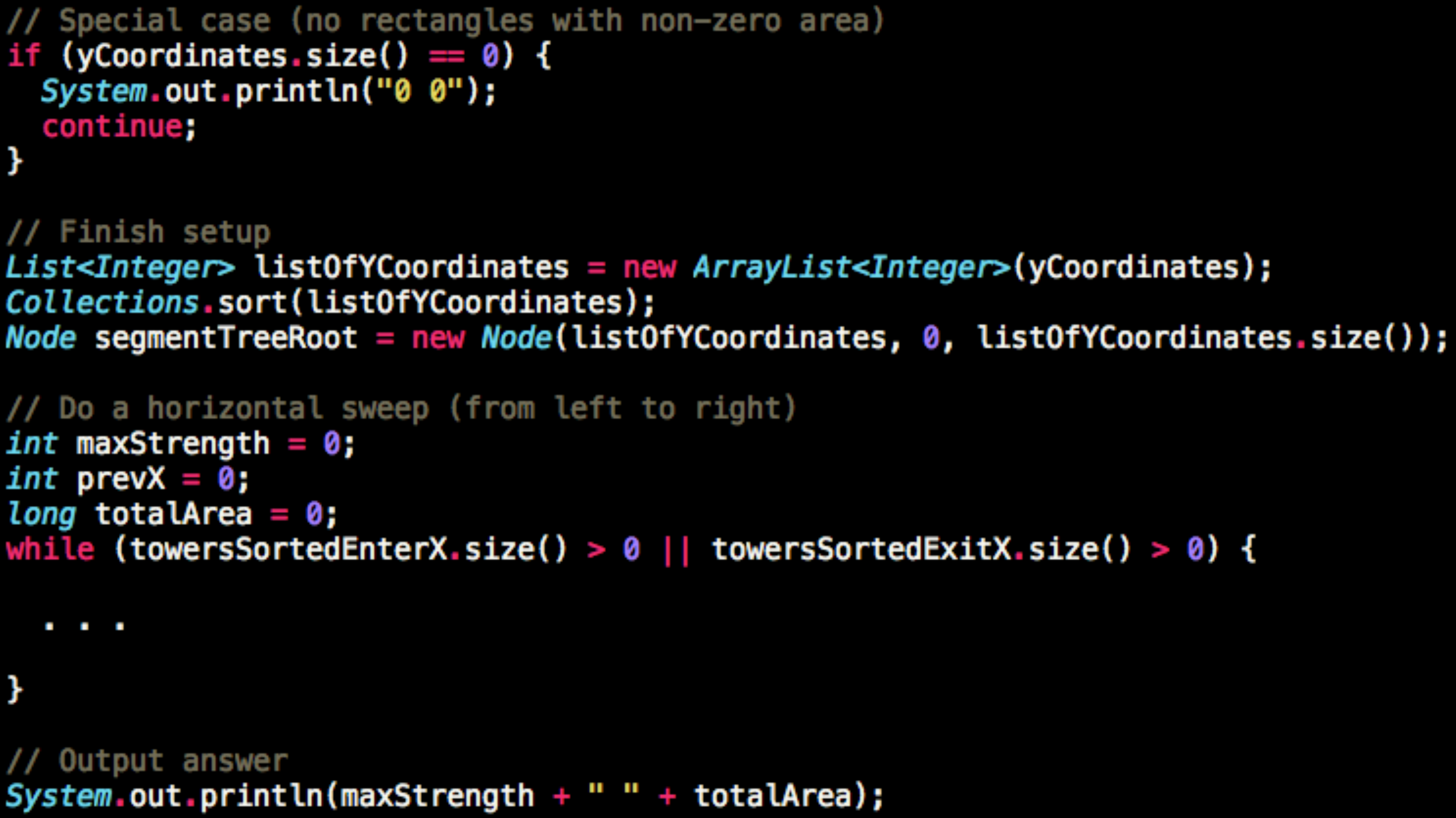


Solution: O(n log(n))



```java
// Process and store input
Queue<Tower> towersSortedEnterX = new PriorityQueue<Tower>(n, new EnterXComparator());
Queue<Tower> towersSortedExitX = new PriorityQueue<Tower>(n, new ExitXComparator());
Set<Integer> yCoordinates = new HashSet<Integer>();
for (int i = 0; i < n; i++) {

  String[] line = br.readLine().split(" ");
  int x1 = Integer.valueOf(line[0]);
  int y1 = Integer.valueOf(line[1]);
  int x2 = Integer.valueOf(line[2]);
  int y2 = Integer.valueOf(line[3]);
  int strength = Integer.valueOf(line[4]);

  Tower tower = new Tower(x1, y1, x2, y2, strength);

  // Ignore towers with 0 area
  if (x1 == x2 || y1 == y2)
    continue;

  towersSortedEnterX.offer(tower);
  towersSortedExitX.offer(tower);
  yCoordinates.add(y1);
  yCoordinates.add(y2);

}
```

Solution: O(n log(n))

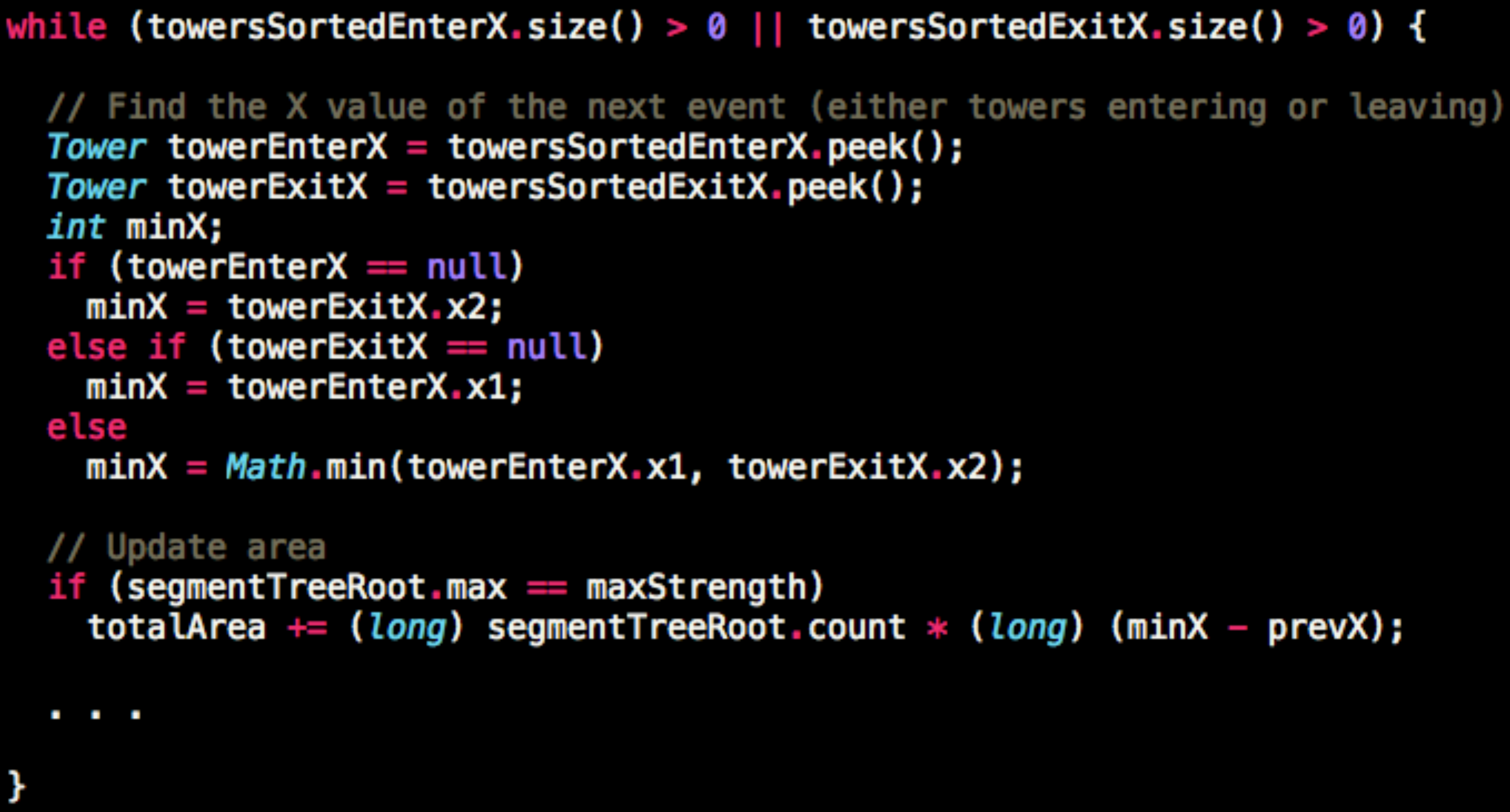


```java
// Special case (no rectangles with non-zero area)
if (yCoordinates.size() == 0) {
  System.out.println("0 0");
  continue;
}

// Finish setup
List<Integer> listOfYCoordinates = new ArrayList<Integer>(yCoordinates);
Collections.sort(listOfYCoordinates);
Node segmentTreeRoot = new Node(listOfYCoordinates, 0, listOfYCoordinates.size());

// Do a horizontal sweep (from left to right)
int maxStrength = 0;
int prevX = 0;
long totalArea = 0;
while (towersSortedEnterX.size() > 0 || towersSortedExitX.size() > 0) {

  . . .

}

// Output answer
System.out.println(maxStrength + " " + totalArea);
```

Solution: O(n log(n))

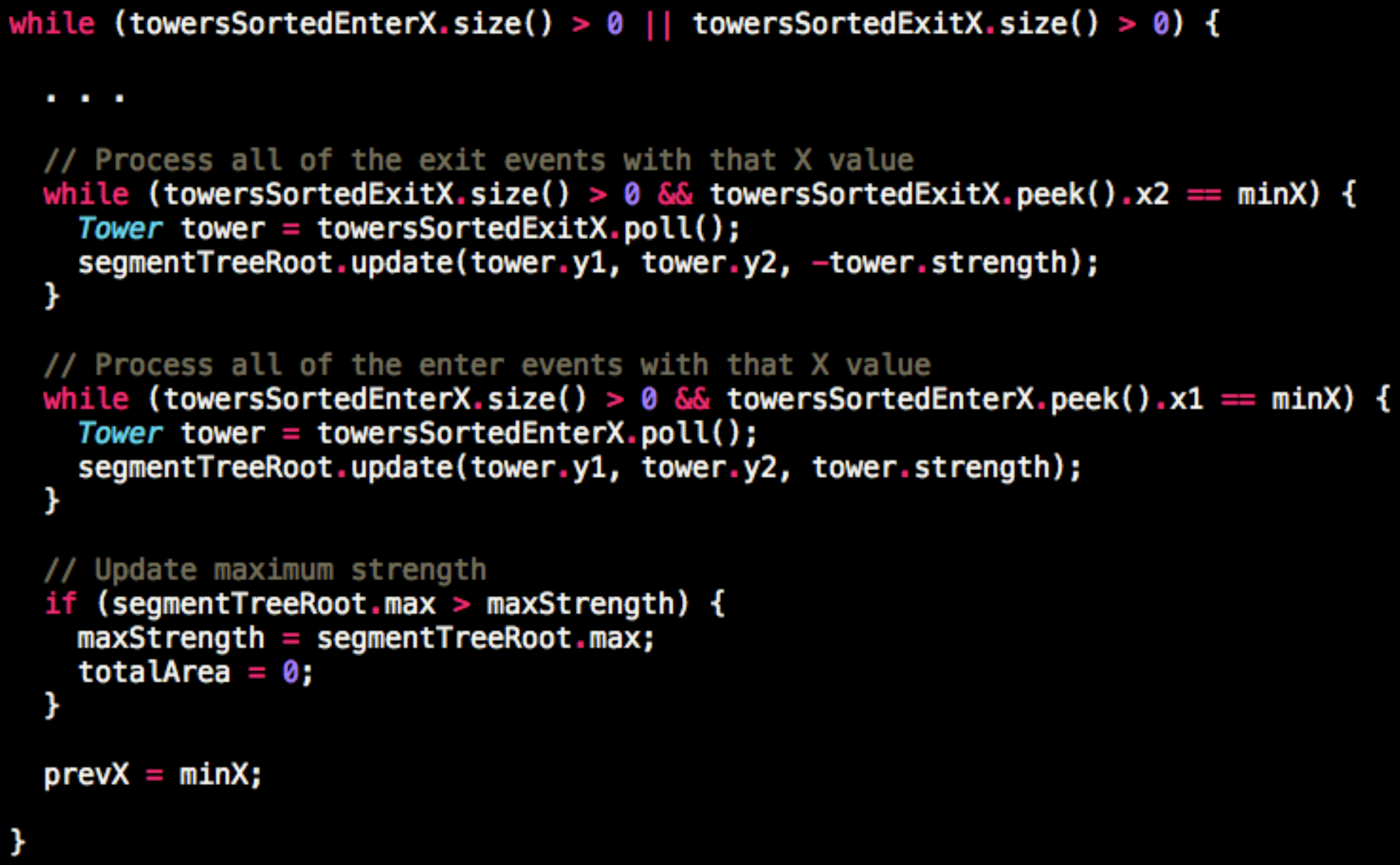
```
// Special case (no rectangles with non-zero area)
if (yCoordinates.size() == 0) {
  System.out.println("0 0");
  continue;
}

// Finish setup
List<Integer> listOfYCoordinates = new ArrayList<Integer>(yCoordinates);
Collections.sort(listOfYCoordinates);
Node segmentTreeRoot = new Node(listOfYCoordinates, 0, listOfYCoordinates.size());

// Do a horizontal sweep (from left to right)
int maxStrength = 0;
int prevX = 0;
long totalArea = 0;
while (towersSortedEnterX.size() > 0 || towersSortedExitX.size() > 0) {

  . . .

}

// Output answer
System.out.println(maxStrength + " " + totalArea);
```

Solution: O(n log(n))



```
while (towersSortedEnterX.size() > 0 || towersSortedExitX.size() > 0) {

  // Find the X value of the next event (either towers entering or leaving)
  Tower towerEnterX = towersSortedEnterX.peek();
  Tower towerExitX = towersSortedExitX.peek();
  int minX;
  if (towerEnterX == null)
    minX = towerExitX.x2;
  else if (towerExitX == null)
    minX = towerEnterX.x1;
  else
    minX = Math.min(towerEnterX.x1, towerExitX.x2);

  // Update area
  if (segmentTreeRoot.max == maxStrength)
    totalArea += (long) segmentTreeRoot.count * (long) (minX - prevX);

  . . .

}
```

Solution: O(n log(n))

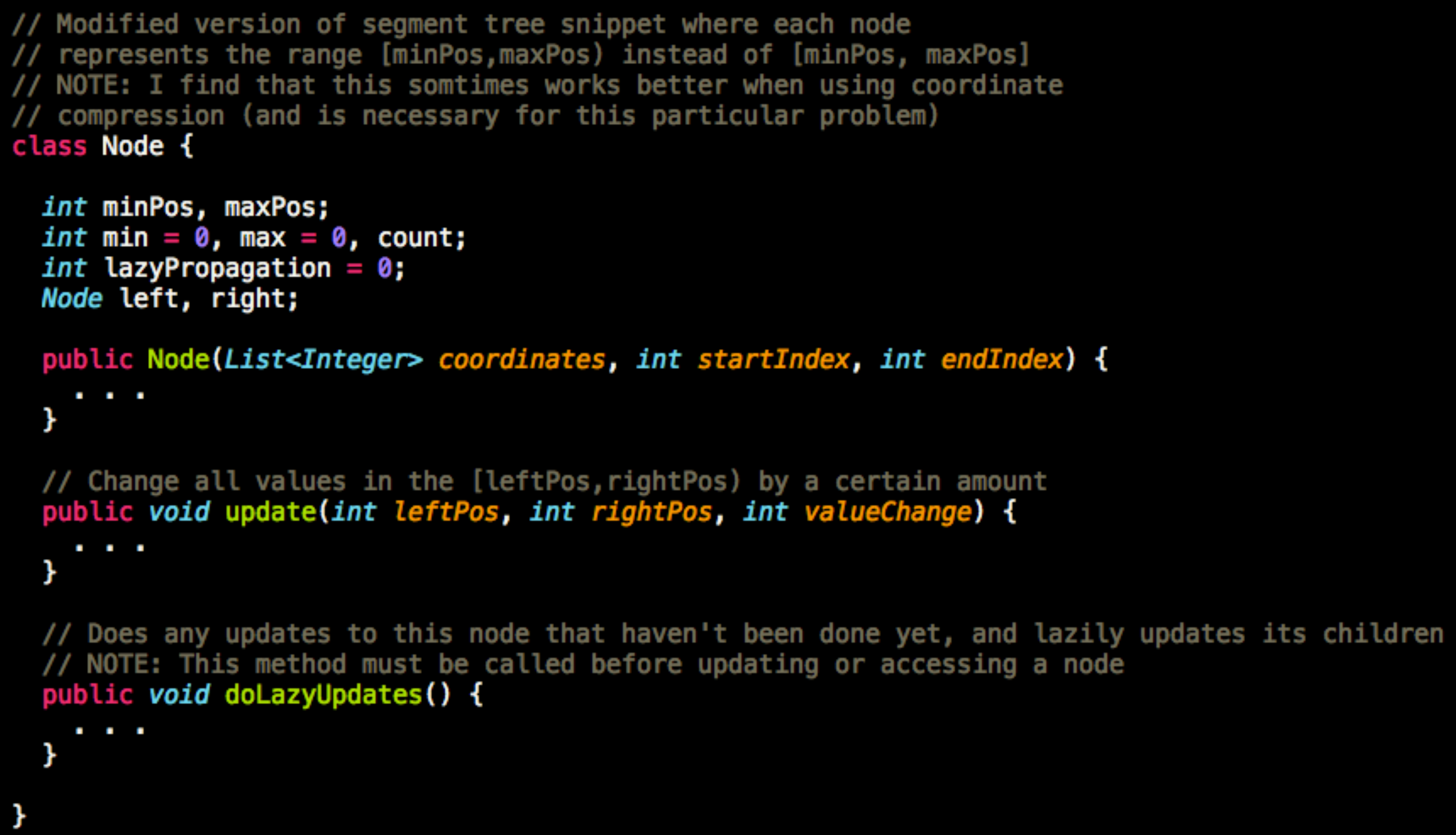


```
while (towersSortedEnterX.size() > 0 || towersSortedExitX.size() > 0) {

  . . .

  // Process all of the exit events with that X value
  while (towersSortedExitX.size() > 0 && towersSortedExitX.peek().x2 == minX) {
    Tower tower = towersSortedExitX.poll();
    segmentTreeRoot.update(tower.y1, tower.y2, -tower.strength);
  }

  // Process all of the enter events with that X value
  while (towersSortedEnterX.size() > 0 && towersSortedEnterX.peek().x1 == minX) {
    Tower tower = towersSortedEnterX.poll();
    segmentTreeRoot.update(tower.y1, tower.y2, tower.strength);
  }

  // Update maximum strength
  if (segmentTreeRoot.max > maxStrength) {
    maxStrength = segmentTreeRoot.max;
    totalArea = 0;
  }

  prevX = minX;

}
```

Solution: O(n log(n))

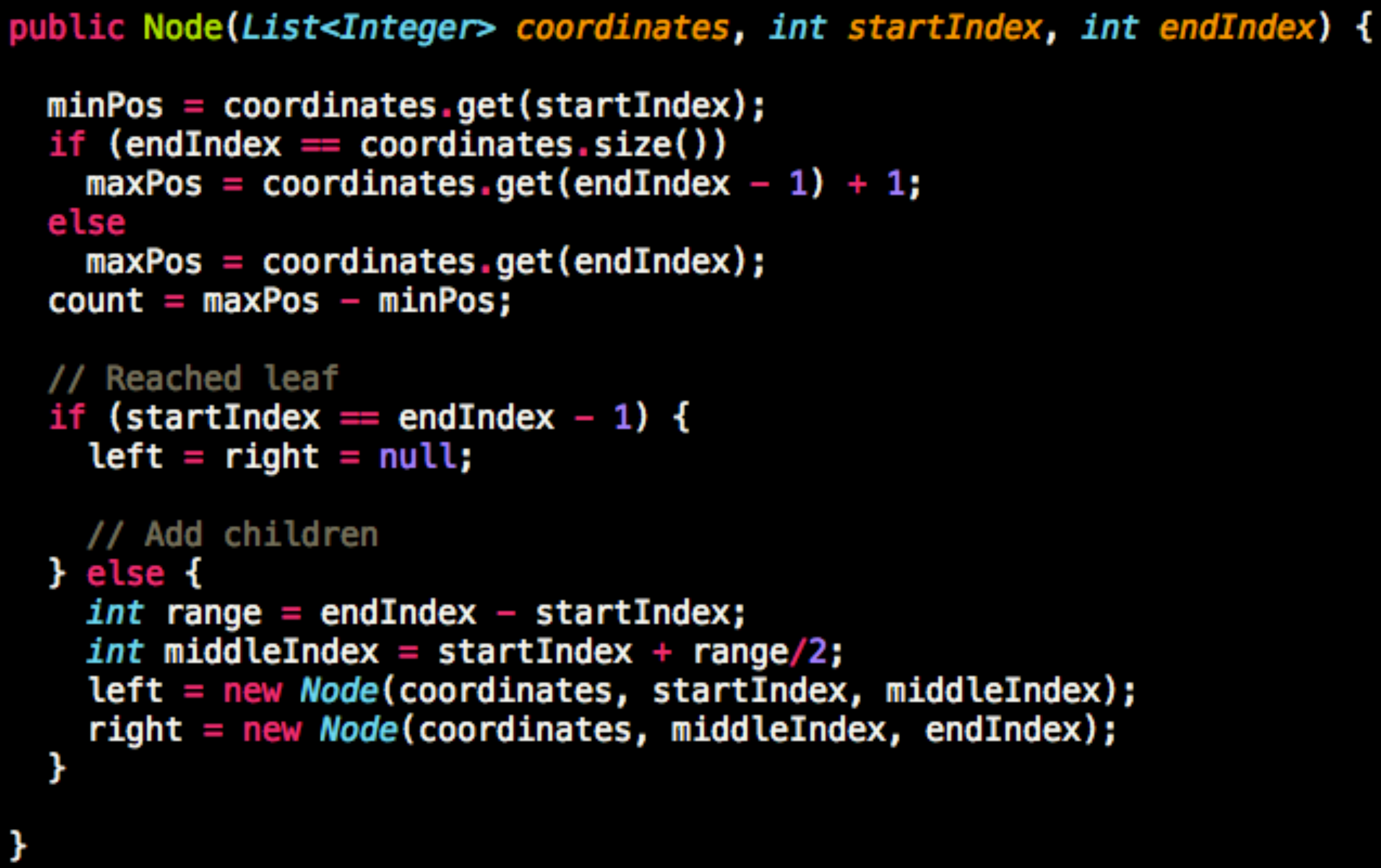
```java
class Tower {

  int x1, y1, x2, y2, strength;

  public Tower(int x1, int y1, int x2, int y2, int strength) {
    this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 = y2; this.strength = strength;
  }

}

// Sort the rectangles by the X value of the left side
class EnterXComparator implements Comparator<Tower> {

  @Override public int compare(Tower a, Tower b) {
    return (new Integer(a.x1)).compareTo(b.x1);
  }

}

// Sort the rectangles by the X value of the right side
class ExitXComparator implements Comparator<Tower> {

  @Override public int compare(Tower a, Tower b) {
    return (new Integer(a.x2)).compareTo(b.x2);
  }

}
```

Solution: O(n log(n))



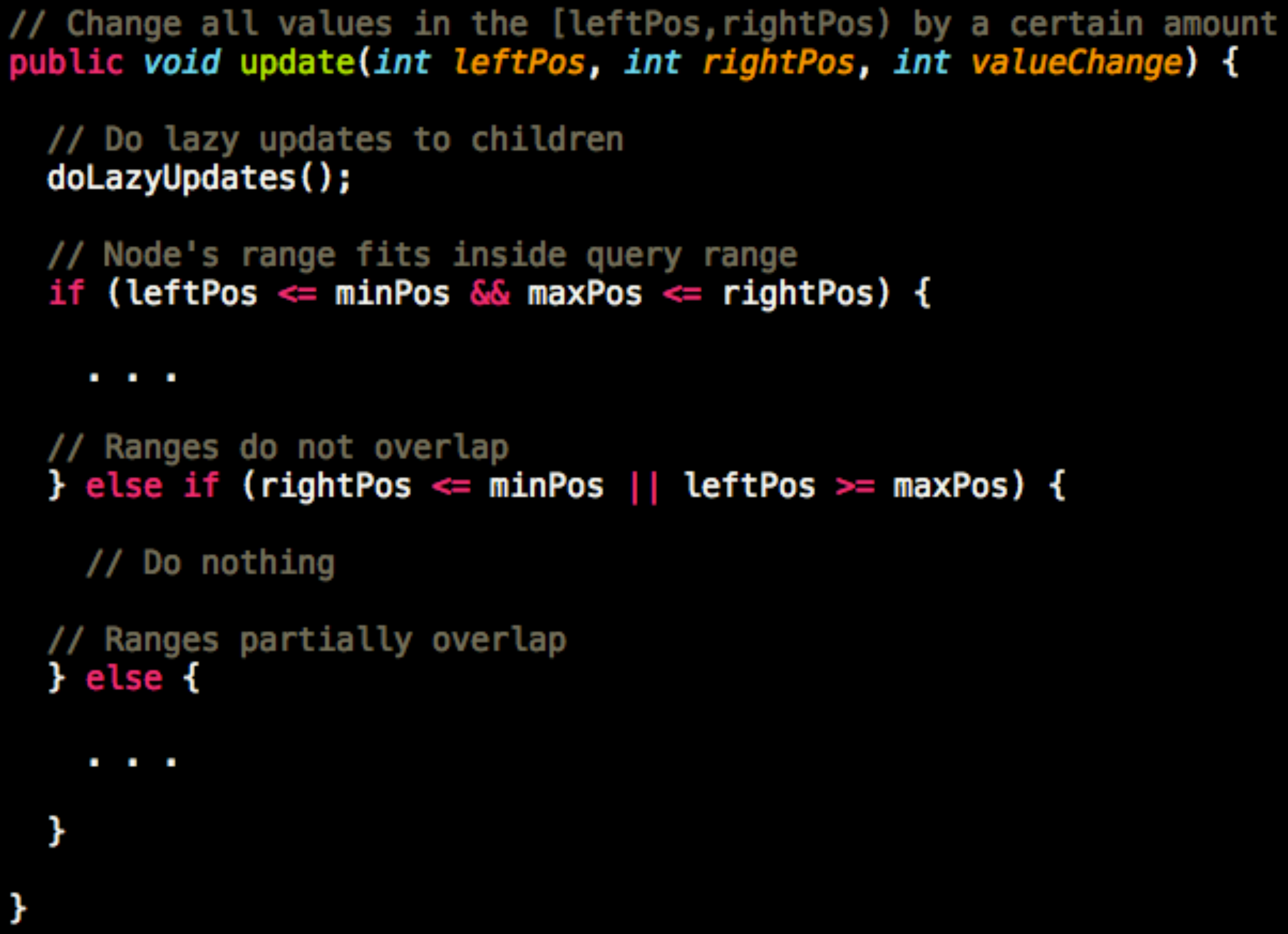
```java
// Modified version of segment tree snippet where each node
// represents the range [minPos,maxPos) instead of [minPos, maxPos]
// NOTE: I find that this somtimes works better when using coordinate
// compression (and is necessary for this particular problem)
class Node {

  int minPos, maxPos;
  int min = 0, max = 0, count;
  int lazyPropagation = 0;
  Node left, right;

  public Node(List<Integer> coordinates, int startIndex, int endIndex) {
    . . .
  }

  // Change all values in the [leftPos,rightPos) by a certain amount
  public void update(int leftPos, int rightPos, int valueChange) {
    . . .
  }

  // Does any updates to this node that haven't been done yet, and lazily updates its children
  // NOTE: This method must be called before updating or accessing a node
  public void doLazyUpdates() {
    . . .
  }

}
```

Solution: O(n log(n))



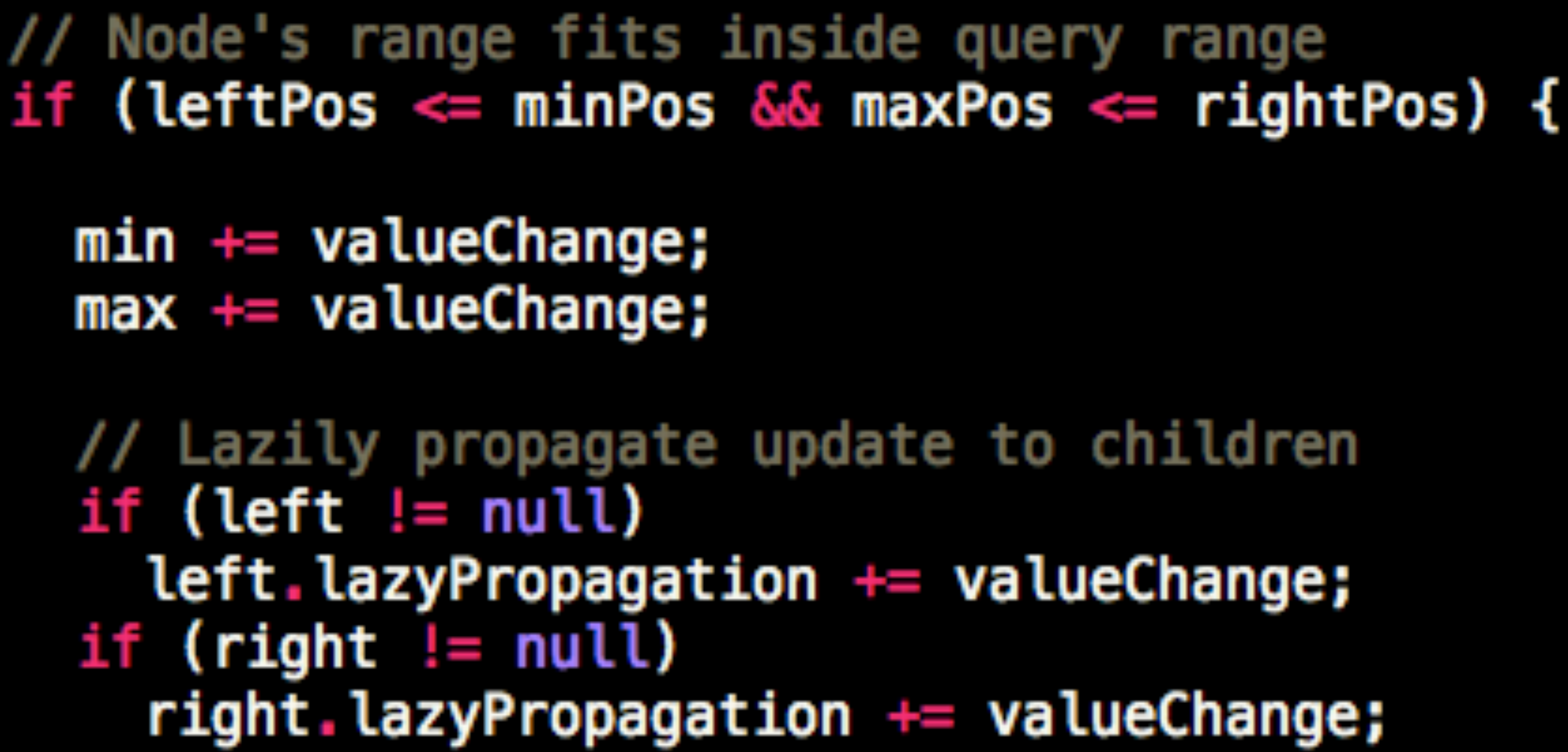
```java
public Node(List<Integer> coordinates, int startIndex, int endIndex) {

  minPos = coordinates.get(startIndex);
  if (endIndex == coordinates.size())
    maxPos = coordinates.get(endIndex - 1) + 1;
  else
    maxPos = coordinates.get(endIndex);
  count = maxPos - minPos;

  // Reached leaf
  if (startIndex == endIndex - 1) {
    left = right = null;

    // Add children
  } else {
    int range = endIndex - startIndex;
    int middleIndex = startIndex + range/2;
    left = new Node(coordinates, startIndex, middleIndex);
    right = new Node(coordinates, middleIndex, endIndex);
  }

}
```

Solution: O(n log(n))



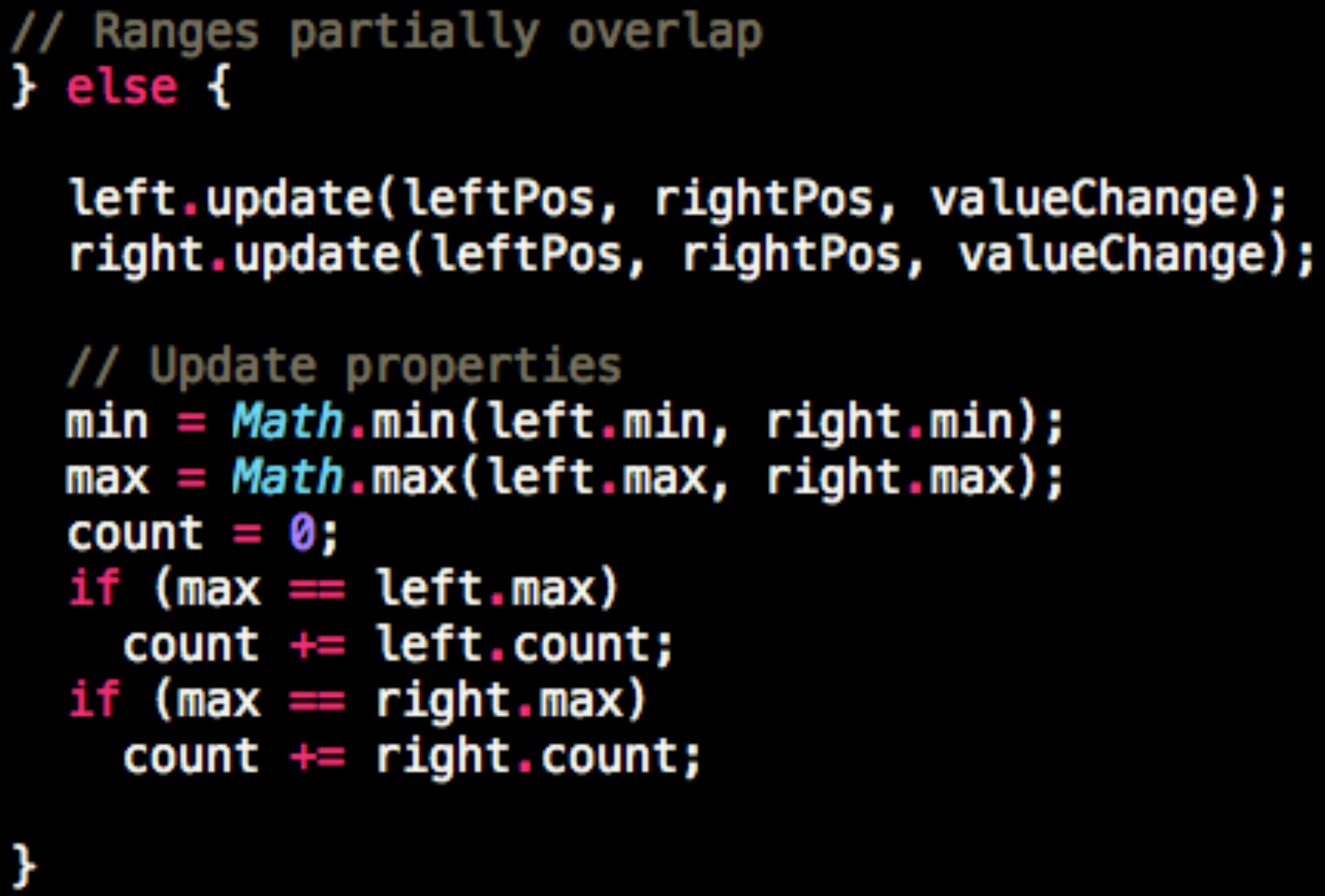
```
// Change all values in the [leftPos,rightPos) by a certain amount
public void update(int leftPos, int rightPos, int valueChange) {

  // Do lazy updates to children
  doLazyUpdates();

  // Node's range fits inside query range
  if (leftPos <= minPos && maxPos <= rightPos) {

    . . .

  // Ranges do not overlap
  } else if (rightPos <= minPos || leftPos >= maxPos) {

    // Do nothing

  // Ranges partially overlap
  } else {

    . . .

  }

}
```

Solution: O(n log(n))



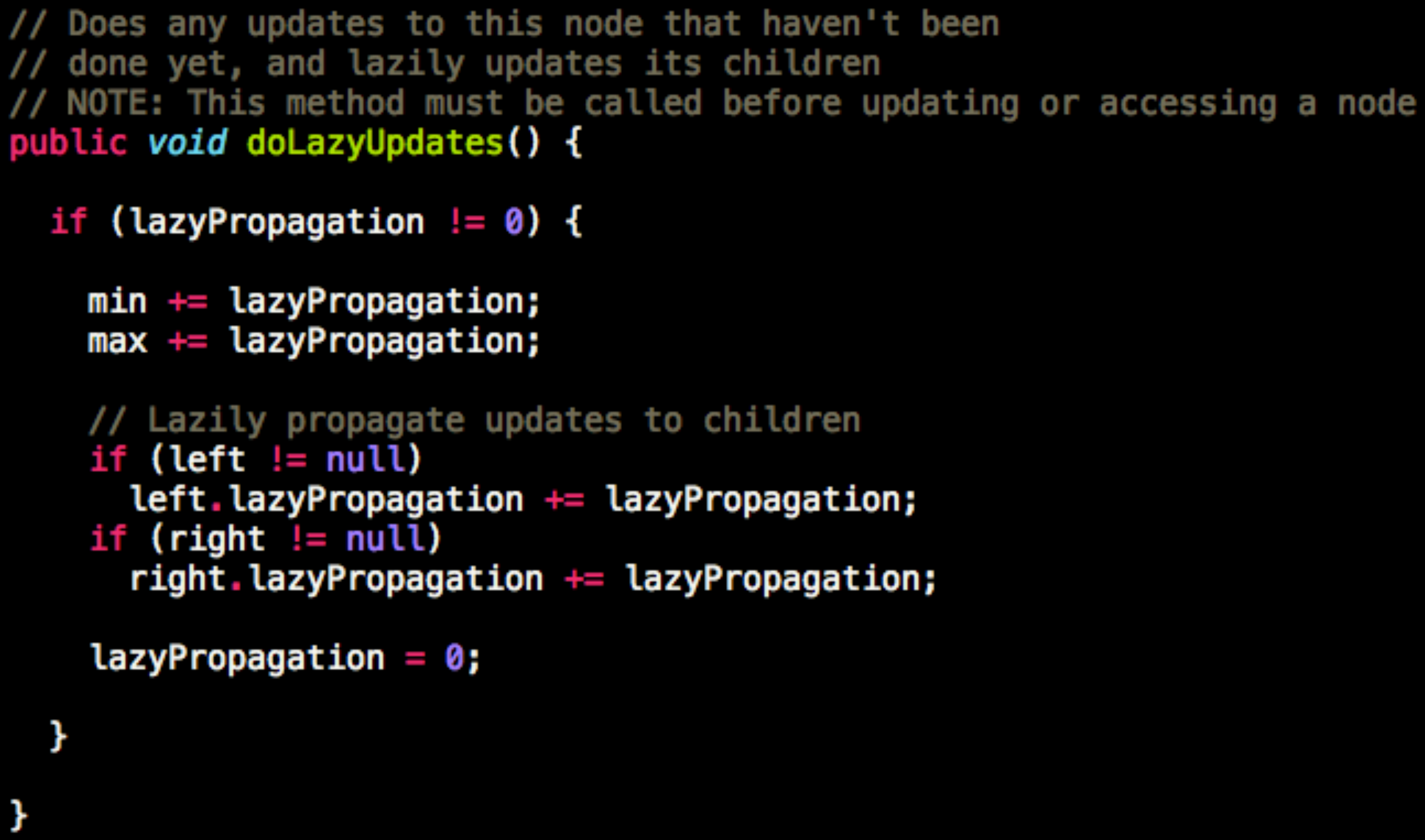
```
// Node's range fits inside query range
if (leftPos <= minPos && maxPos <= rightPos) {

  min += valueChange;
  max += valueChange;

  // Lazily propagate update to children
  if (left != null)
    left.lazyPropagation += valueChange;
  if (right != null)
    right.lazyPropagation += valueChange;
```

Solution: O(n log(n))



```
// Ranges partially overlap
} else {

  left.update(leftPos, rightPos, valueChange);
  right.update(leftPos, rightPos, valueChange);

  // Update properties
  min = Math.min(left.min, right.min);
  max = Math.max(left.max, right.max);
  count = 0;
  if (max == left.max)
    count += left.count;
  if (max == right.max)
    count += right.count;

}
```

Solution: O(n log(n))



```
// Does any updates to this node that haven't been
// done yet, and lazily updates its children
// NOTE: This method must be called before updating or accessing a node
public void doLazyUpdates() {

  if (lazyPropagation != 0) {

    min += lazyPropagation;
    max += lazyPropagation;

    // Lazily propagate updates to children
    if (left != null)
      left.lazyPropagation += lazyPropagation;
    if (right != null)
      right.lazyPropagation += lazyPropagation;

    lazyPropagation = 0;

  }

}
```

Conclusion

- Not an algorithm, but a technique
- Can be applied to a wide variety of problems
- Powerful technique that can lead to innovative and efficient solutions - typically O(n log(n))

Further Problems

- https://open.kattis.com/problems/gridmst
- http://community.topcoder.com/stat?c=problem_statement&pm=4463&rd=6536
- http://community.topcoder.com/stat?c=problem_statement&pm=4559&rd=7225

Sources

- https://en.wikipedia.org/wiki/Sweep_line_algorithm
- https://www.topcoder.com/community/data-science/data-science-tutorials/line-sweep-algorithms/
- http://contest.felk.cvut.cz/09prg/solved/solution2009.ppt
- http://www.comp.nus.edu.sg/~acmicpc/rcdreport-preliminarycontest.pdf