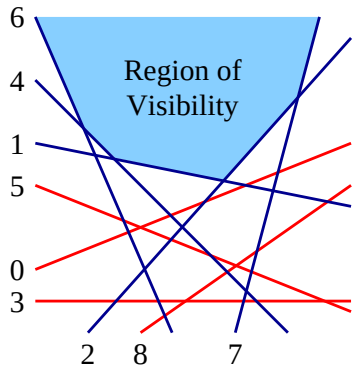
CS29003 Algorithms Laboratory

Assignment No: 3

Date: 28–January–2019

Divide-and-Conquer Algorithms

You are given n non-vertical lines $L_0, L_1, L_2, \ldots, L_{n-1}$. Each line L_i is specified by two floating-point (`double`) numbers m_i and c_i. The equation of L_i is $y = m_i x + c_i$ (a vertical line cannot be so represented). You are sitting at the point $(0, \infty)$, that is, infinitely above all finite points in the plane. The lines are assumed to be opaque. As you look down, you can see a portion of the plane delimited by portions of the lines. Your task is to find out your region of visibility as defined by the lines. The following figure illustrates this concept. In this assignment, assume that the lines are in *general position*, that is, no two of the given lines are parallel, and no three of the given lines are concurrent.



Part 1: Data types

Define a data type to store a line (two `double` values m and c). You may additionally require to store the line number (an integer in the range $[0, n-1]$) in each line structure. You have to work with points of intersection of lines, so define an additional data type to store a point in the two-dimensional plane (again two `double` values x and y).

Part 2: The First Algorithm

A little bit of geometric insight suggests that the steepest line with negative slope and the steepest line with positive slope must constitute the first and the last portions of the region of visibility. In between them, you have finite segments of some other lines. Implement the following algorithm (given in pseudocode) in a function `method1` for the determination of the region of visibility.

1. Find the line with the smallest slope. Mark this as the current line K. Also initialize the current point to $Q = (-\infty, \cdot)$.
2. So long as it is possible to add another line, repeat the following steps.
   (a) Let $K = L_i$ be the current line.
   (b) Find the points of intersection P_{ij} of L_i with all other lines L_j which are not already included in the boundary of the region of visibility (maintain an array of visibility flags).
   (c) Do not consider any P_{ij} if $x(P_{ij}) < x(Q)$ (where Q is the current point). If no such j exists, you are done, so break the loop.
   (d) Find the intersection point P_{ij^*} for which $x(P_{ij^*})$ is smallest among all P_{ij} with $x(P_{ij}) > x(Q)$.
   (e) Add, to the boundary of the region of visibility, the segment of L_{j^*} from the current point Q to the intersection point P_{ij^*}.
   (f) Update the current line K to L_{j^*}, and the current point Q to P_{ij^*}.

Evidently, this algorithm has a worst-case running time of $O(n^2)$, because it expends $O(n)$ effort for discovering each segment on the boundary of the region of visibility.

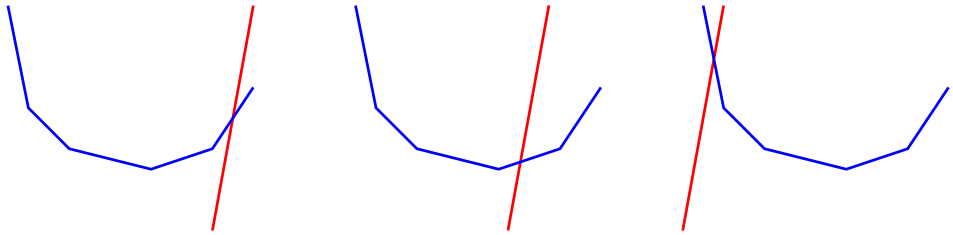
Part 3: Precomputation for the Second Algorithm

We want to come up with an $O(n \log n)$-time algorithm for the visibility problem. This algorithm requires the input lines be sorted in the increasing order of their slopes. Implement merge sort for this purpose. Assume that no two lines are parallel, so there is no duplicate slope among the input lines. You should write your own merge-sort function. We need a worst-case $O(n \log n)$-time sorting algorithm for the given problem, so do not implement any worst-case $\Theta(n^2)$-time sorting algorithm.

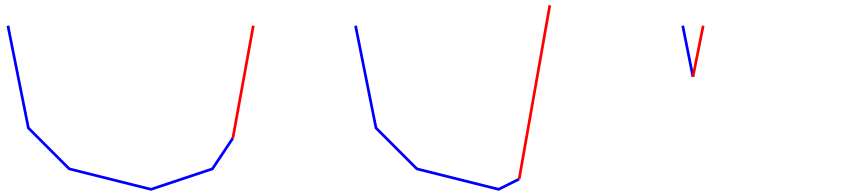
Part 4: The Second Algorithm

An $O(n \log n)$-time algorithm is now sketched. In view of Part 3, we assume that the n lines are sorted in the increasing order of their slopes. This has already taken $O(n \log n)$ time. The rest of the algorithm finishes in $O(n)$ time only.

Consider the lines in the sorted order. The line with the smallest slope must be the first part of the boundary of the region of visibility. Suppose that $L_{i_1}, L_{i_2}, \ldots, L_{i_k}$ are the lines appearing in that sequence in the current boundary. You now consider the next line L_j from the sorted list. Depending upon the position of the next line L_j, the boundary changes to $L_{i_1}, L_{i_2}, \ldots, L_{i_t}, L_j$ for some t in the range $1 \leqslant t \leqslant k$. The following figure demonstrates this incremental construction.



Change of the boundary of the region of visibility
for three different positions of the next line



(a) (b) (c)

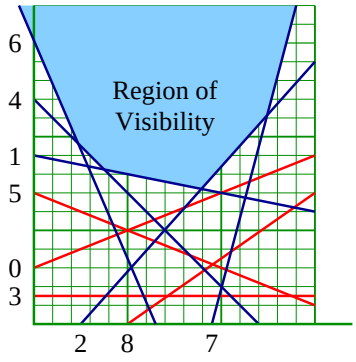
Figure out how you determine t (that is, the segments that you need to throw for incorporating the next line). Use a stack to store the growing and shrinking boundary (you may use an STL stack). Write a function **`method2`** to implement this algorithm. The function should run in $O(n)$ time in the worst case.

The *main*() function

- Read n followed by n pairs (m_i, c_i) for $i = 0, 1, 2, \ldots, n-1$. Store the lines in an array. Print the lines in the format given in the sample output.
- Call **`method1`** to compute and print the boundary of visibility.
- Sort the lines by your merge-sort function. Print the lines in the sorted order.
- Call **`method2`** to compute and print the boundary of visibility.

Sample output

The output given below corresponds to the lines in the figure of the first page. This figure is repeated below with coordinate lines shown. The bottom left corner is the origin $(0,0)$. Larger sample(s) can be found at the lab website.

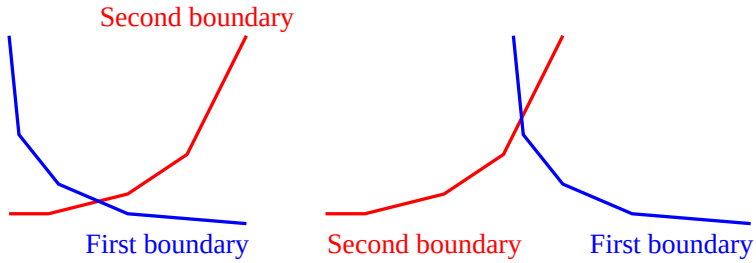



```
n = 9

     0.4000000000      3.0000000000
    -0.2000000000      9.0000000000
     1.1200000000     -2.8000000000
     0.0000000000      1.5000000000
    -1.0000000000     12.0000000000
    -0.4000000000      7.0000000000
    -2.3076923077     15.0000000000
     3.7500000000    -35.6250000000
     0.7000000000     -3.5000000000

+++ Lines before sorting
Line     0: y =       0.4000000000 x +     3.0000000000
Line     1: y =      -0.2000000000 x +     9.0000000000
Line     2: y =       1.1200000000 x -     2.8000000000
Line     3: y =       0.0000000000 x +     1.5000000000
Line     4: y =      -1.0000000000 x +    12.0000000000
Line     5: y =      -0.4000000000 x +     7.0000000000
Line     6: y =      -2.3076923077 x +    15.0000000000
Line     7: y =       3.7500000000 x -    35.6250000000
Line     8: y =       0.7000000000 x -     3.5000000000

+++ Method 1
Line     6: From MINUS_INFINITY to (2.2941176471,9.7058823529)
Line     4: From (2.2941176471,9.7058823529) to (3.7500000000,8.2500000000)
Line     1: From (3.7500000000,8.2500000000) to (8.9393939394,7.2121212121)
Line     2: From (8.9393939394,7.2121212121) to (12.4809885932,11.1787072243)
Line     7: From (12.4809885932,11.1787072243) to PLUS_INFINITY

+++ Lines after sorting
Line     6: y =      -2.3076923077 x +    15.0000000000
Line     4: y =      -1.0000000000 x +    12.0000000000
Line     5: y =      -0.4000000000 x +     7.0000000000
Line     1: y =      -0.2000000000 x +     9.0000000000
Line     3: y =       0.0000000000 x +     1.5000000000
Line     0: y =       0.4000000000 x +     3.0000000000
Line     8: y =       0.7000000000 x -     3.5000000000
Line     2: y =       1.1200000000 x -     2.8000000000
Line     7: y =       3.7500000000 x -    35.6250000000

+++ Method 2
Line     6: From MINUS_INFINITY to (2.2941176471,9.7058823529)
Line     4: From (2.2941176471,9.7058823529) to (3.7500000000,8.2500000000)
Line     1: From (3.7500000000,8.2500000000) to (8.9393939394,7.2121212121)
Line     2: From (8.9393939394,7.2121212121) to (12.4809885932,11.1787072243)
Line     7: From (12.4809885932,11.1787072243) to PLUS_INFINITY
```

Submit a single C/C++ source file. Do not use global/static variables.

A theoretical exercise

Here is a divide-and-conquer algorithm that does not use merge sort. Use a linear-time algorithm to find the line with the median slope value. Partition the set of lines with respect to this median-slope line into two (almost) equal-sized halves. Recursively construct the regions of visibility of the two halves. Propose an $O(n)$-time algorithm to merge the two recursively constructed regions into one. The following figure illustrates that the merging procedure first needs to locate the intersecting segments in the two recursively computed boundaries.



(a) (b)

Two illustrations of merging

The running time of this algorithm satisfies $T(n) = 2T(n/2) + O(n)$. By the master theorem, we have $T(n) = O(n \log n)$.