Design and Analysis of Algorithms, Assignment 5

As usual, each part of this assignment is submitted as a separate file. As you finish up the parts, you can turn them in at `https://submit.ncsu.edu/`. Before the deadline, you can submit updated parts of any section whenever you want. They'll automatically replace your previous submission.

Be sure you use the right filename for each part of the assignment, including the right capitalization, and don't pack multiple files into an archive for submission. You'll get a small penalty if we have to rename files or unpack them from an archive before we grade them (or just to tell what you've submitted).

For programming problems, you'll need to submit an implementation that compiles and runs using the instructions from the previous assignment. For programs, you'll generally be expected to read input from standard input and write output to standard output. Try to match our expected output format exactly, without printing extra output (like prompts) that the assignment didn't ask for.

1. Line-Sweep Algorithm for Line Segment Intersection (40 points)

   You're going to write a program, `sweep.c` (or `.cpp`, `.java`, `.py`). This program will implement the line-sweep algorithm for finding an intersecting pair among a large set of line segments.

   Like we did when we were describing the algorithm in class, you can assume no line segments are perfectly vertical, and that no more than two line segments intersect at exactly the same point.

   ## Input

   You'll read input from standard in. It will start with a positive integer, n. This will be followed by n lines, each giving four real numbers, the X, Y coordinates of one endpoint of a line segment, then the X, Y coordinates of the other endpoint.

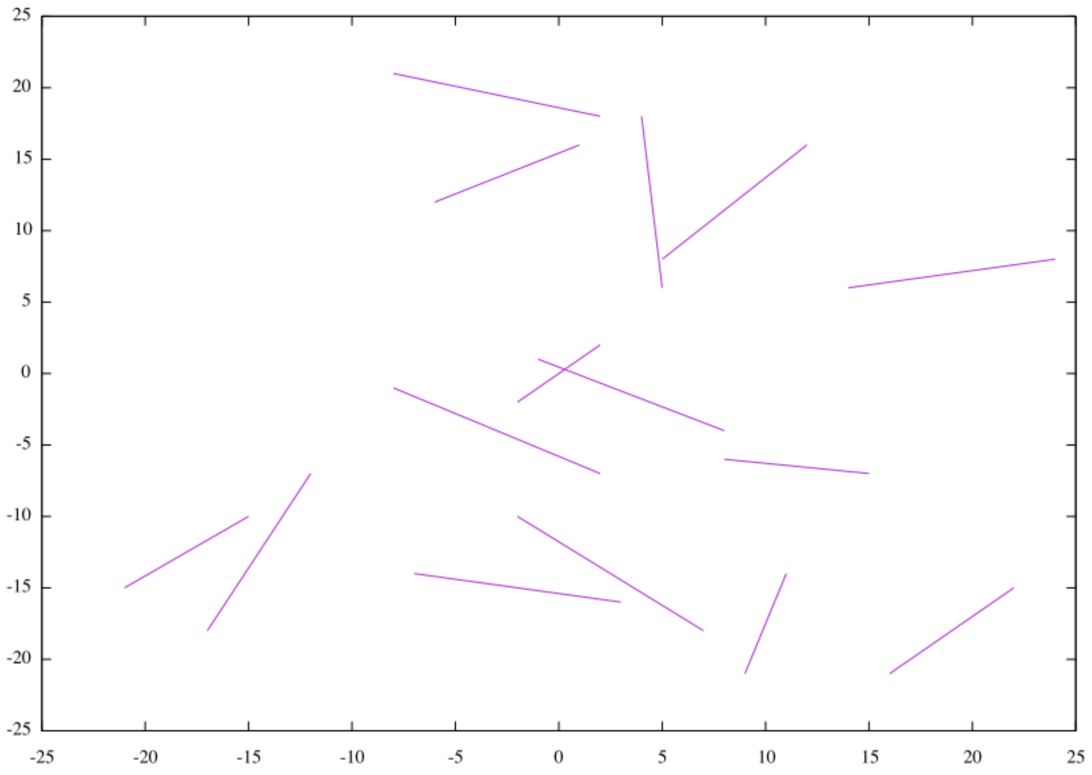
   ## Output

   As output, print a line containing the X and Y coordinates of the point where two line segments intersect. Round the coordinates to two fractional digits of precision, and put one space between them.

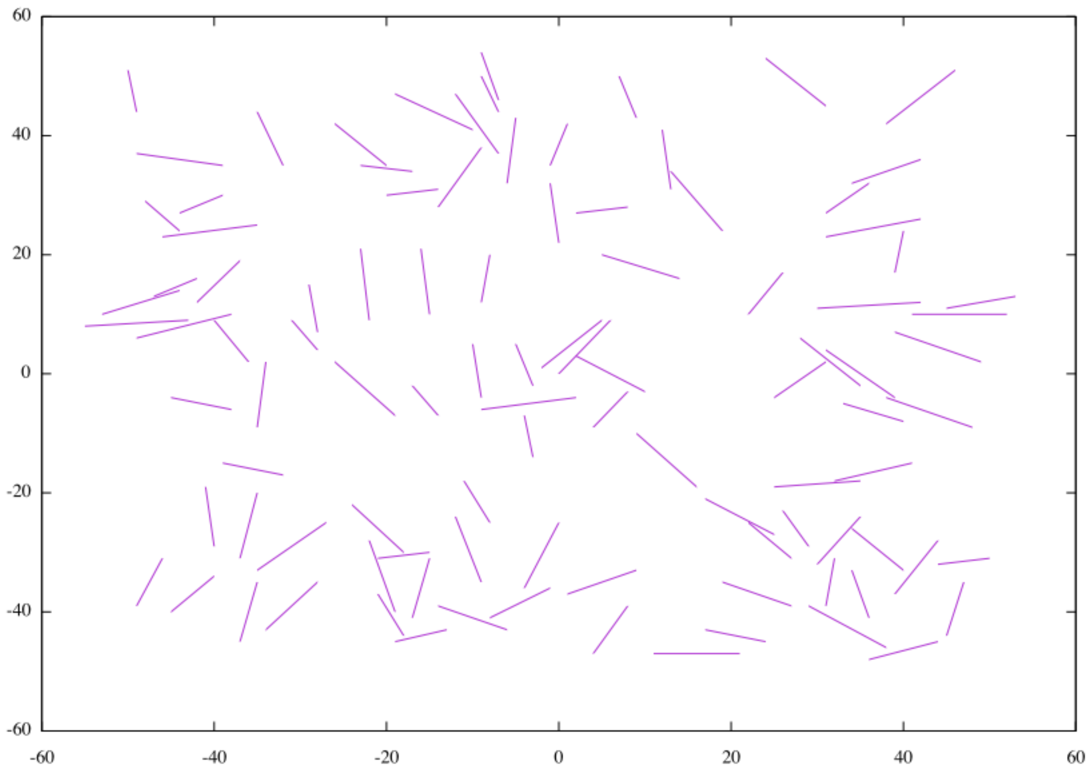
   For all the inputs, there should just be one pair of line segments that intersect.

   ## Input Files

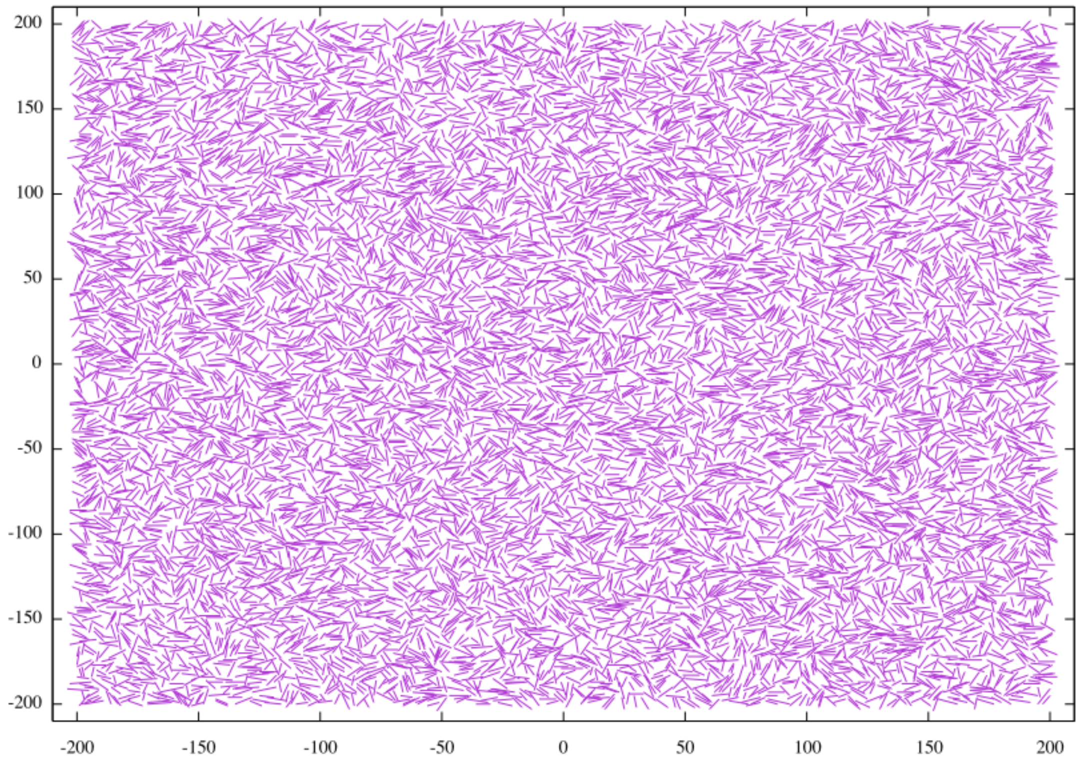
   I'm providing 3 test inputs for this problem. The first one is small, with just 15 line segments. They look like this. You can see the one intersection point near the middle of the figure.



The next test case has 100 line segments. Here, again, the intersecting pair are near the center. They meet right on the endpoint of one of the line segments.



The third test input contains 10,000 line segments (but just one intersection). Can you find it? Sounds like a good job for a program.



Sample Execution

Once your program is working, you should be able to run it on my sample input files as follows. I'm providing expected output files. Since the inputs just have one pair of intersecting lines, your output should match the expected output exactly.

```
# Run on the smallest sample input
$ python sweep.py < input_s1.txt
0.29 0.29
```

Extra Credit

If you'd like 10 points of extra credit, have your program and report all intersections between line segments, instead of just the first one encountered. You should still be able to do this in $O(n \log n)$ time, as long as there aren't more than $O(n \log n)$ intersections.

Print out all the intersections sorted by X coordinate (so, you'll have to find them all before printing them), one intersection per line. If two intersections have the same X coordinate, print the ones with lower Y coordinates first.

2. KMP and Performance Evaluation (40 points)

   Let's compare string searching algorithms, including the standard string search mechanism for your chosen programming language. Write a program called kmp.c (or kmp.cpp, kmp.java, kmp.py). This will compare the performance of the naive, $O(nm)$ string search algorithm, the KMP string search algorithm and the standard string search technique for your language. We'll say the standard way of doing a string search is the strstr() function if you're in C. If you're programming in C++, we'll say it's the find() method on stl::string. In Java, we'll say it's the indexOf() method on String, and in Python, we'll say it's the find() method on string.

String Input

Your program will take one optional command-line argument giving the name of a file. This file should contain two lines, the first giving the text you're supposed to search through (the haystack) and the second containing the pattern your supposed to look for (the needle). So, for example, if the program is run as follows (in python):

```
python kmp.py input_kmp1.txt
```

It will use the first two lines from the file 'input_kmp1.txt' as its haystack and needle strings.

If no command-line argument is given, your program should use needle and haystack strings designed by you. Try to choose these strings so that KMP performs noticeably better than the other two string search techniques. These will need to be large strings if we're going to see a performance difference, so you'll definitely want to write some looping code to create these strings (rather than just trying to type them in).

For example, running the program like the following should cause it to use the strings you designed (if it's in Java).

```
java kmp
```

String Search

In your program, define two of your own functions to perform string searching. The first will be called naive(). It will take two parameters, the haystack string then the needle string. The naive algorithm will use the naive, $O(nm)$ algorithm to find the first occurrence of needle in the haystack and return the integer index of the first character of its occurrence. If it doesn't occur, your function will return -1.

You'll also implement a function called kmp() that takes the same two parameters and returns the same value as naive(). Internally, kmp() will use the Knuth-Morris-Pratt algorithm to find the needle int he haystack.

Performance Comparison

When your program is run, it will use all three algorithms to perform the string search. First it will report on the position of the needle reported by naive() and the runtime of the naive string search in milliseconds.[1] See the sample execution for the expected output format.

Then, use the standard built-in technique for finding a substring in a large string (described above). Report the position where it finds the needle (or -1) and the running time of this string search.

Finally, use your KMP algorithm to do a string search and report where it finds the substring and how long it takes. For all three techniques, the location where it finds the first occurrence of needle should be the same (but, this will be good to test with), but the runtimes will probably be different.

[1]Just measure the cost of the string search itself, not the time required to read the strings from the input file or to generate them if no input file is given.

Sample Execution

If you run your program as follows (if it's written in C or C++), it will report the results of the three string search techniques and their runtimes of the sample input file. Here, the input strings are too small for string searching to take more than a millisecond of execution time:

```
$ ./kmp input_kmp1.txt
found at 60
naive search time: 0
found at 60
standard search time: 0
found at 60
kmp search time: 0
```

If you run your program as follows (if it's written in Java), it will create a large string you designed to try to show a performance advantage for KMP. As you can see, for my program, the needle occurs at position 788124 in the haystack (so, obviously, I'm using large strings). For the strings I created, the built-in string search is faster than my naive solution, but KMP is even faster.

```
$ java kmp
found at: 788124
naive search time: 3401
found at: 788124
standard search time: 524
found at: 788124
kmp search time: 20
```

Performance Report

In a file called, `report.pdf`, give a report explaining the strings you used to try to exhibit better performance from KMP compared to the other two string search techniques. Show the output of your program and briefly explain your results and any conclusions you can draw from them. This should take about half a page.

Submitting your Work

When you're done, submit your source code and your performance report to the Homework_5 submission locker on wolfware classic.