Computer Science I

Exam 1 - Fall 2025

School of Computing
College of Engineering
University of Nebraska-Lincoln

This is the first exam for Computer Science I (CSCE 155E and CSCE 155H) for the fall 2025 semester. It has two parts and **both** are required.

- Absolutely **no collaboration is allowed with any individual**
- The use of any AI tools is strictly prohibited, all work must be your own
- Regular help hours will be held, but LAs will not give the same level of help as they would on a hack/lab
- For those in the *HONORS* section: you must do one program in C and one in Java (the choice is yours). For Java, name your source/classes `Point.java` and `Battery.java` respectively and place them in the `unl.soc` package.
- Good luck!

Part 1 [25 points]

This part is worth 25 points with the same expectations and point distributions as a regular Hack.

Problem Statement

Consider a point (x, y) in the cartesian plane. The point may lie in one of four *quadrants* as depicted in the figure below. Alternatively, if the point lies on the `x-axis`, `y-axis`, or at the `Origin` then it does not lie in one of the quadrants.

In addition, there are two notions of distance from the origin, $(0, 0)$ to the point (x, y); one is the *Manhattan distance* which is calculated as:

$$|x| + |y|$$

which measures the distance you'd have to travel horizontally/vertically to reach (x, y). The second notion is the usual euclidean distance calculated as:

$$\sqrt{x^2 + y^2}$$

Write a program that reads in x and y and outputs *where* the point is located (with respect to quadrant or axis) and the two *distances* from the origin.

For example, if we invoke your program from the command line using

```
./a.out 2.4 3.9
```

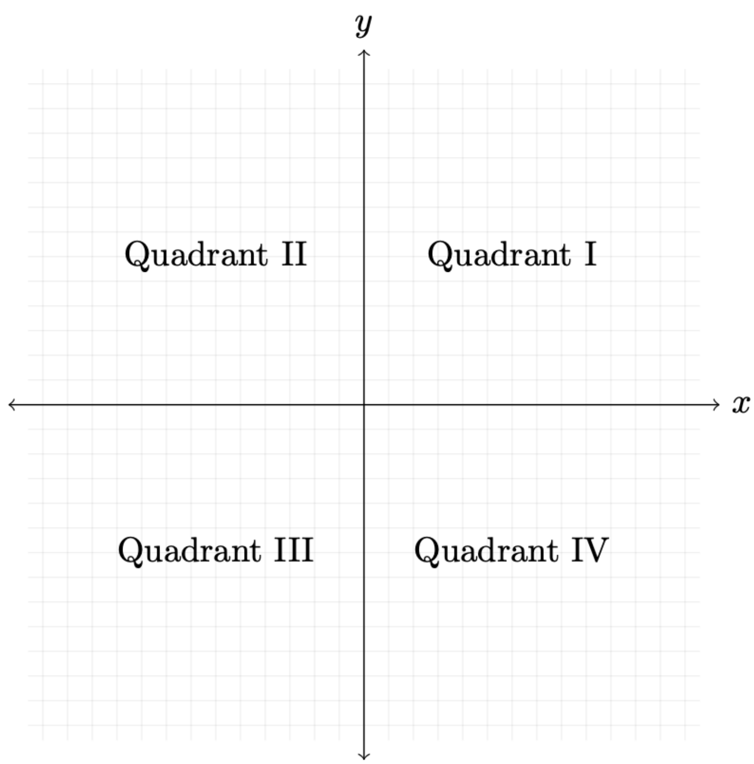

Figure 1: Quadrants of the Cartesian Plane

Figure 1: Quadrants

it would correspond to the point $(2.4, 3.9)$ and your output should look something like the following.

```
Point: (2.400000, 3.900000)
Location:           Quadrant I
Manhattan Distance: 6.300000
Euclidean Distance: 4.579301
```

Instructions

- Place your code in a source file named `point.c` and turn it in through codepost.
- You should perform some basic error checking and input validation, exiting with an appropriate message that includes they keyword `ERROR`
- Verify all the tests pass; you may resubmit as many times as you like until the due date.
- We will grade on style, documentation, design, and correctness just like a Hack.

Part 2 [50 points]

Problem Statement

Consider a fully charged cell phone batter (100% charge). There are several different ways that one can model the drain on the battery over time (t measured in hours) until the battery reaches 0% charge.

- A linear model assumes constant usage and drains the battery by 10% each hour:
$$B_1(t) = 100 - 10t$$
- A *piecewise* model will assume that the battery is drained at different rates; it will drain quicker when fully charged, then will entry power saving mode as time goes on:
$$B_2(t) = \begin{cases} 100 - 20t & \text{for } 0 \leq t < 2 \\ 60 - 5(t-2) & \text{for } 2 \leq t < 5 \\ 45 - 2(t-5) & \text{for } 5 \leq t \leq 14 \end{cases}$$
For example, if it has been $t = 4$ hours then $B_2(4) = 60 - 5(4-2) = 50$ (or 50% remaining)
- An exponential model assumes that usage will drop off over time:
$$B_3(t) = 100 \cdot e^{-0.2t}$$
- A more advanced model uses a “smart” app to save the battery over time.
$$B_4(t) = \frac{100}{1 + e^{0.8(t-5)}}$$

For all four models, if the battery reaches zero it will remain as zero (the formula values could become negative, but a negative charge does not make sense).

Write a program that takes, as command line arguments, an ending time t in hours and an increment (in fractional hours) and assumes that the charge starts at 100% and produces a table that outputs the expected charge (percentage) for each model at each time increment.

For example, if we ran your program with $t = 5$ and an increment of .25:

```
./a.out 5 .25
```

Your table should look **something** like the following.

```
Time          B1        B2        B3        B4
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
   0.00    100.00    100.00    100.00     98.20
   0.25     97.50     95.00     95.12     97.81
   0.50     95.00     90.00     90.48     97.34
   0.75     92.50     85.00     86.07     96.77
   1.00     90.00     80.00     81.87     96.08
   1.25     87.50     75.00     77.88     95.26
   1.50     85.00     70.00     74.08     94.27
   1.75     82.50     65.00     70.47     93.09
   2.00     80.00     60.00     67.03     91.68
   2.25     77.50     58.75     63.76     90.02
   2.50     75.00     57.50     60.65     88.08
   2.75     72.50     56.25     57.69     85.81
   3.00     70.00     55.00     54.88     83.20
   3.25     67.50     53.75     52.20     80.22
   3.50     65.00     52.50     49.66     76.85
   3.75     62.50     51.25     47.24     73.11
   4.00     60.00     50.00     44.93     69.00
   4.25     57.50     48.75     42.74     64.57
   4.50     55.00     47.50     40.66     59.87
   4.75     52.50     46.25     38.67     54.98
   5.00     50.00     45.00     36.79     50.00
```

Instructions

- Place your code in the source file `battery.c`.
- Turn your solution in using codepost.
- Verify all the tests pass; you may resubmit as many times as you like until the due date.
- You should perform some rudimentary error checking and input validation, exiting with an appropriate message that includes `ERROR`
- We will grade on style, documentation, design, and correctness just like a Hack. However, the point values will be doubled.