Art+Logic Programming Challenge (parts 2 & 3)

NOTE: A few things to keep in mind when completing this part of the challenge:

- In Part 1, our reviewers were instructed to give some leeway when reviewing submissions; we wanted you to be able to quickly write a solution and submit it without worrying too much about writing production-quality code. When reviewing Part 2 submissions, that goes away. Please use this as an opportunity to show what kind of code you'd write in a production environment.
- We also pointed you at our programming style guide which you can read at http://styleguide.artandlogic.com, but said that we weren't expecting submitted code to necessarily follow it. On this part, we're not looking for complete adherence to Art+Logic style, but we are looking for a good-faith attempt to write code in Art+Logic house style.

The Problem

Okay, we'll start with the decoding function that you wrote as part of your Part 1 submission.

Assume that we've been hired by a client to develop a system that uses data values encoded in the format that we outlined in part 1, so if you were given text like

```
400000007F7F
```

...your code will be able to correctly convert it into the corresponding list of integer values

```
[0, -8192, 8191]
```

The data we're working with will describe a set of simple commands to build a vector-based drawing system. This system uses a pen-based model where an imaginary pen can be raised or lowered. If the pen is moved while it is down, we draw along the line of motion in the current color. If the pen is moved while it is up, no drawing is done.

The commands supported in this mini-language are:

- Clear the drawing and reset all parameters
- Raise/lower the pen
- Change the current color
- Move the pen

In this system, commands are represented in the data stream by a single (un-encoded) opcode byte that can be identified by always having its most significant bit set, followed by zero or more bytes containing encoded data values.

Any unrecognized commands encountered in an input stream should be ignored.

Generating Output

For the purposes of this challenge, we'd rather avoid the complications of actually generating graphics output. Instead, we'll just define a simple text output that shows that each command in the data stream is being interpreted correctly. The format for the output of each command type is shown here as the last row in each table below.

Commands

Clear

Command	`CLR`
Opcode	`F0`
Parameters	(none)
Output	`CLR;\n`

Cause the drawing to reset itself, including:

- setting the pen state to up
- setting the pen position back to (0,0)
- setting the current color to (0, 0, 0, 255) (black)
- clearing any output on the screen (not shown in our example output here)

Pen Up/Down

Command	`PEN`
Opcode	`80`
Parameters	0 = pen up any other value = pen down
Output	either `PEN UP;\n` or `PEN DOWN;\n`

Change the state of the pen object to either up or down. When a pen is up, moving it leaves no trace on the drawing. When the pen is down and moves, it draws a line in the current color.

Set Color

Command	`CO`
Opcode	`A0`
Parameters	`R G B A` Red, Green, Blue, Alpha values, each in the range 0..255. All four values are required.
Output	`CO {r} {g} {b} {a};\n` (where each of the r/g/b/a values are formatted as integers 0..255)

Change the current color (including alpha) of the pen. The color change takes effect the next time the pen is moved. After clearing a drawing with the `CLR;` command, the current color is reset to black (0,0,0,255).

Move Pen

Command	`MV`
Opcode	`C0`
Parameters	`dx0 dy0` *`[dx1 dy1 .. dx`$_n$ `dy`$_n$`]`* Any number of (dx,dy) pairs.
Output	if pen is up, move the pen to the final coordinate position as defined below If pen is down: `MV (`x_o`, `y_0`) (`x_1`, `y_1`)` *`[... (`x_n`, `y_n`)]`*`;\n`

Change the location of the pen relative to its current location. If the pen is down, draw in the current color. If multiple (x, y) points are provided as parameters, each point moved to becomes the new current location in turn.

Also note that the values used in your output should be absolute coordinates in the drawing space, not the relative coordinates used in the encoded movement commands.

For example, after clearing a drawing, the current location is the origin at (0, 0). If the pen is moved (10, 10) and then (5, -5), the final coordinate position of the pen will be at (15, 5). For the purposes of this exercise, the string output would be

```
MV (10, 10) (15, 5);
```

if the pen is down, but just

```
MV (15, 5);
```

if the pen is up.

If the specified motion takes the pen outside the allowed bounds of (-8192, -8192) .. (8191, 8191), the pen should move until it crosses that boundary and then lift. When additional movement commands bring the pen back into the valid coordinate space, the pen should be placed down at the boundary and draw to the next position in the data file.

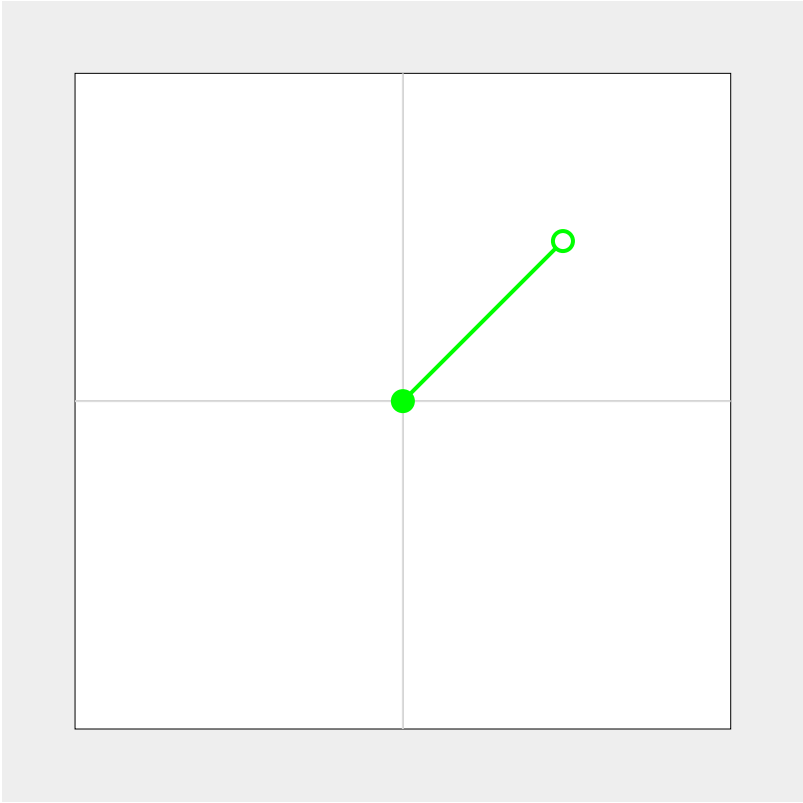
Examples

Simple Green Line

Input Data:

```
F0A04000417F4000417FC040004000804001C05F205F20804000
```

Set color to green, draw a line from (0,0) to (4000, 4000). Filled circle in this diagram indicates pen down position, empty circle indicates pen up position.



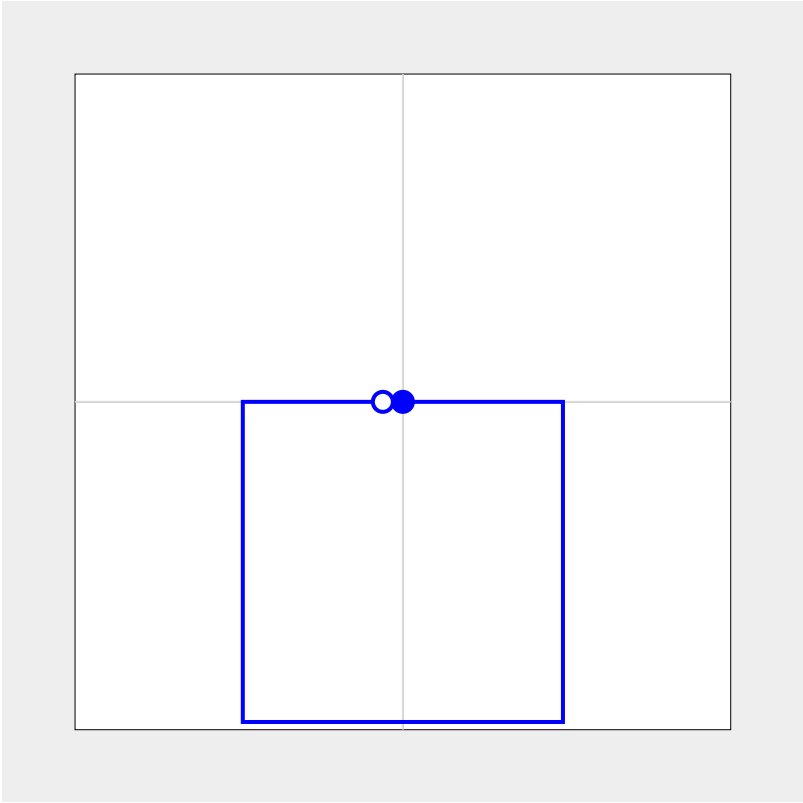
Output:

```
CLR;
CO 0 255 0 255;
MV (0, 0);
PEN DOWN;
MV (4000, 4000);
PEN UP;
```

Blue Square

Input Data:

```
F0A040004000417F417FC040004000904000047684F5057384000804001C0
5F204000400001400140400040007E405B2C4000804000
```



Output:

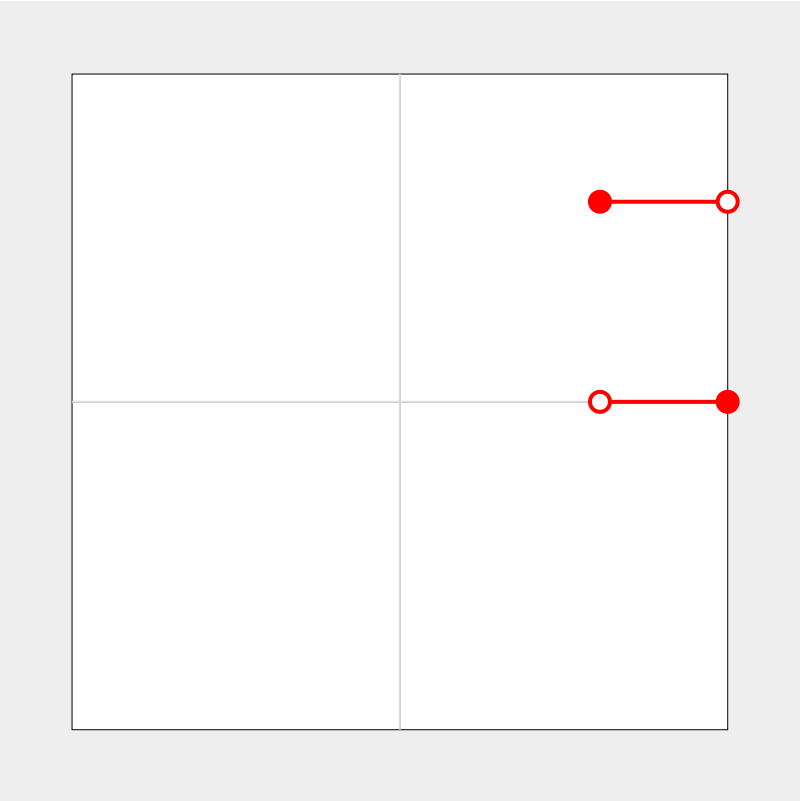
```
CLR;
CO 0 0 255 255;
MV (0, 0);
PEN DOWN;
MV (4000, 0) (4000, -8000) (-4000, -8000) (-4000, 0) (-500, 0);
PEN UP;
```

Clipping at the Edge of the Drawing

When the drawing stays inside the boundaries of the coordinate space we're working with, things are simple -- there's a 1:1 correspondence between the commands in the input and the output to be generated by your program. Since the movement commands in the input stream are each relative to the current location of the pen, it's possible for that current position to move outside of the valid coordinates, and your program will need to include some special logic to handle that situation intelligently.

Input Data:

```
F0A0417F40004000417FC067086708804001C06708400040001878187840008040000
```



Output:

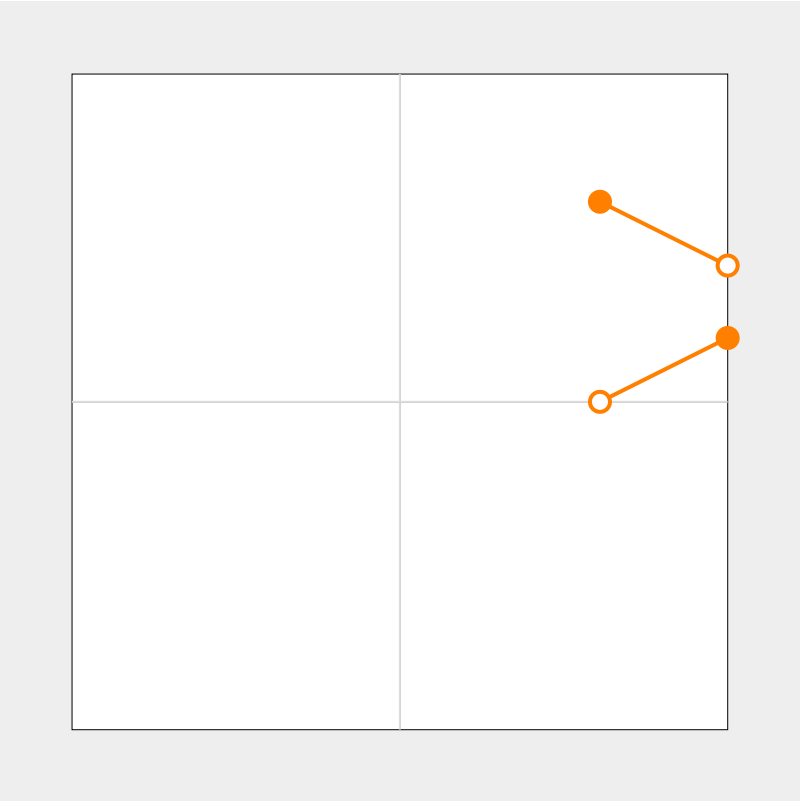
```
CLR;
CO 255 0 0 255;
MV (5000, 5000);
PEN DOWN;
MV (8191, 5000);
PEN UP;
MV (8191, 0);
PEN DOWN;
MV (5000, 0);
PEN UP;
```

Note that when the pen moves outside of the legal coordinate space, we lift the pen where the line goes out of bounds and then move to the point where the line re-enters our space before putting the pen back down again.

Here's a (slightly) more interesting example:

Input Data

```
F0A0417F41004000417FC067086708804001C067082C3C18782C3C804000
```



Output:

```
CLR;
CO 255 128 0 255;
MV (5000, 5000);
PEN DOWN;
MV (8191, 3405);
PEN UP;
MV (8191, 1596);
PEN DOWN;
MV (5000, 0);
PEN UP;
```

Your Task

As in Part 1, supply source code (and whatever additional collateral files may be useful or needed for us to validate and test) a working program that accepts a string of hexadecimal data encoded as outlined in this document and and converts those commands into to a block of text that follows the text format detailed above.

This program can be a command line application that reads from stdin or a file and writes to stdout or a file, or could be a web page or mobile/desktop app that uses a text edit box to accept input values by typing or pasting (or loading a file) and a button to convert that input into the desired output format.

Your goal here isn't just to write code that does the barest of minimum work to barf out a correct answer; we're looking to get a taste of what your habits and instincts are for writing production-quality code. As we noted in Part 1, we'd prefer that you limit the use of any third-party libraries or frameworks where possible—remember that the goal here is to help us see how awesome your code can be when you're working on a problem where none of those libraries exist yet.

As in Part 1, your program must be written using one of our preferred programming languages:

- C++
- C#
- JavaScript
- Java *(see below)*
- Objective-C
- Python

- Swift

Please don't submit solutions in Java unless you are specifically asking to be considered as an Android developer.

Successful applicants typically take between 6 and 12 hours to complete this part of the challenge.

Please do *not* include any executable, `.jar` or other `.zip` files inside your submission. For security reasons, our systems will reject any submissions that do include any of these files types without notifying either of us. Complete information on the prohibited file types can be found at https://support.google.com/mail/answer/6590?hl=en.

Note that one of the prohibited file extensions is `.js` — if your solution contains JavaScript source, please rename these files to use the extension `.javascript` to ensure that your submission reaches us.

Please package your source and any other files accompanying your submission into a ZIP file named `firstName_lastName_Part2.zip`.

Part 3

Also please create a file in `.doc`, `.pdf`, or plain text format named `firstName_lastName_Part3.{doc|pdf|txt}` that contains your answers to these questions:

1. Tell us about something that you recently learned (or are currently learning). How do you approach learning a new skill?
2. Describe the differences between projects you've been involved with (or seen) that have succeeded, and those that have not been successful.
3. The languages, tools, and technologies that we use as developers are always changing. What specific languages/frameworks/platforms should developers be learning now to be ready for the next few years?
4. Discuss the most effective team you've been a part of. What made it so? What was your role on the team?
5. What specific approaches or techniques do you use in your code to make sure that it's bug-free and performant?

Submitting Your Solution

To submit your ALPC part 2 and 3 solution to us, please complete the form at http://www.artandlogic.com/alpc2, being sure to upload both

- `firstName_lastName_Part2.zip`
- `firstName_lastName_Part3.txt`

If you have questions, email recruiting@artandlogic.com.

File version 2019-08-01