Eric Roberts
CS 106J

Handout #37
May 15, 2017

Assignment #5—The Enigma Machine

Due: Friday, May 26, 5:00 P.M.
Note: This assignment may be done in pairs

Your task in this assignment is to write a program that simulates the operation of the Enigma machine, which was used by the German military to encrypt messages during World War II. This handout focuses on what you have to do to complete the assignment. Additional background on the Enigma machine—and how the codebreakers at Bletchley Park managed to break it—appears in Handout #36.

The starter files

When you download the **Assignment5** folder from the web, the folder includes a file called `Enigma.js` that contains a main program to get you started. When you run the program we've given you, it produces the screen display shown in Figure 1.

Figure 1. The output produced by the starter version of Enigma.js

Sadly, even though the screen display shows the entire Enigma machine, that doesn't mean that you're finished with the assignment. The display is simply a **GImage** that shows what the Enigma machine looks like when viewed from the top. The display produced by the starter program is completely inert. Your job is to make it interactive.

The code for the starter file appears in Figure 2. The first three lines read the image file **EnigmaTopView.png** into a **GImage**, create a **GWindow** object that is the same size as that image, and then add the image to the window. That code is all you need to display the image shown in Figure 1.

The next line in the main program is

```
runEnigmaSimulation(gw);
```

which calls **runEnigmaSimulation**, which you have to write. As the comments indicate, your implementation of this function must perform the following operations:

1. Create a JavaScript object that encapsulates the state of the Enigma machine.
2. Create graphical objects that sit on top of the image, overlaying the keys and lamps.
3. Adds listeners to the graphics window that forward mouse events to these objects.

Each of these operations is described in a subsequent section.

Figure 2. The supplied starter code for Enigma.js

```
/*
 * File: Enigma.js
 * ---------------
 * This program implements a graphical simulation of the Enigma machine.
 */

import "graphics";
import "EnigmaConstants.js";

/* Main program */

function Enigma() {
   var enigmaImage = GImage("EnigmaTopView.png");
   var gw = GWindow(enigmaImage.getWidth(), enigmaImage.getHeight());
   gw.add(enigmaImage);
   runEnigmaSimulation(gw);
}

// You are responsible for filling in the rest of the code.  Your
// implementation of runEnigmaSimulation should perform the
// following operations:
//
// 1. Create an object that encapsulates the state of the Enigma machine.
// 2. Create and add graphical objects that sit on top of the image.
// 3. Add listeners that forward mouse events to those objects.

function runEnigmaSimulation(gw) {
   // Fill this in, along with helper functions that decompose the work
}
```

The starter folder also includes two other files. The **EnigmaTopView.png** file is the image of the Enigma machine used for the background. The **EnigmaConstants.js** file contains definitions for all the constants used in the simulator. The contents of **EnigmaConstants.js** are presented piece-by-piece at the points in the handout at which the individual constants apply.

Creating an object to encapsulate the state of the Enigma machine

The primary goal of this assignment is to give you practice working with JavaScript arrays and objects, at least in their simple form as aggregates used to combine individual data items into a single structure. To make sure that different parts of the Enigma simulator can interact, it is useful to define a single variable that includes all the information the rest of the program needs. Our solution calls that variable **enigma** and initializes it like this inside the **runEnigmaSimulation** function:

```
var enigma = { . . . field definitions containing the information . . . };
```

What you need to do over the course of the assignment is figure out what information you need and how to represent it. That understanding, however, will only come as you complete the other parts of the assignment and come to recognize what information you need to share among the different components of the program.

As a starting point, however, you can simply write the following declaration, which defines **enigma** as an object containing no fields at all:

```
var enigma = { };
```

As you complete later parts of the assignment, you can go back and add fields between the curly braces as you learn what data you need. Code that acts as a placeholder for a more detailed implementation you supply later is called a ***stub.***

Creating the graphical objects

Each of the graphical components initialized by **runEnigmaSimulation** is a **GObject** that sits on top of the Enigma image. For example, one of the graphical objects you will need to create looks like this:

That object is a **GCompound** consisting of a light gray **GOval**, a slightly smaller dark gray **GOval** sitting on top of it, and a **GLabel** containing the letter **Q**. When you add that **GCompound** to the graphics window, you place it on top of the background image so that it occupies the same position as the **Q** key on the image.

One of the advantages of adding a **GCompound** on top of the image is that you can change the properties of the components that make up the **GCompound**. For example, you can mark that a key is down by changing the color of its **GLabel** to **KEY_DOWN_COLOR**,

which is defined in the `EnigmaConstants.js` file as the hexadecimal string `"#CC3333"`. After you change the color of the label, the `GCompound` for the **Q** key looks like this:

Forwarding mouse events to the graphical objects

Another advantage of creating `GCompound` objects and putting them on top of the image is that doing so makes it easier to respond to mouse events. The basic idea is that if a mouse event occurs whose location is inside a `GCompound`, the mouse listener for the window can call a method in the `GCompound` to respond. In computer science, this strategy is called ***forwarding.*** The listener for the `GWindow` detects the event but transfers responsibility for responding to the object at which that event occurred.

Implementing the forwarding strategy requires two steps, as follows:

1. As part of the code for `runEnigmaSimulation`, add a listener for each type of mouse event you need to detect anywhere in the Enigma application. Keys on the Enigma keyboard, for example, need to respond to the `"mousedown"` event, so you will need to include a listener for that event. The callback function then calls `getElementAt` to see what `GObject` exists at that location.[1] If `getElementAt` returns a non-`null` value, the listener can then check to see whether that object defines a function to implement a response. If so, the listener can simply call that function. If not, the listener simply ignores that event. To ensure, however, that the function has access to the entire data structure for the Enigma machine, it is essential to pass the `enigma` variable as a parameter. This technique is illustrated by the following code, which implements the `"mousedown"` action using this event-forwarding model:

```
var mousedownAction = function(e) {
   var obj = gw.getElementAt(e.getX(), e.getY());
   if (obj !== null && obj.mousedownAction !== undefined) {
      obj.mousedownAction(enigma);
   }
};
gw.addEventListener("mousedown", mousedownAction);
```

2. Add a field directly to the `GCompound` whose value is a function that performs whatever action is necessary when the mouse event occurs. For example, if the variable `key` holds the `GCompound` representing one of the keys, you can add a `mousedownAction` function like this:

```
key.mousedownAction = function(enigma) {
   Take whatever action is necessary in response.
}
```

[1] As it happens, `getElementAt` will always return an object in this application because the `GImage` covers the entire window. Even so, it is safer to include explicit tests for `null` in the callback function in case you want to reuse this code in other applications that might not include an object that covers everything.

Milestone #1: Create the keyboard

The first step on your way to implementing an interactive Enigma machine is to add the `GCompound` objects that overlay the keys. As described in the preceding section, the **Q** key is an overlapping combination of three shapes, as follows:

The constants that define the appearance of a key appear in Figure 3, which contains the relevant lines from `EnigmaConstants.js`.

It is, of course, insufficient to create a `GCompound` just for the **Q** key. What you need to do instead is create 26 such objects, one for each letter. Moreover, since cutting and pasting 26 copies of the same code is both tedious and error prone, you want to perform this operation using a loop that cycles through the letters from **A** to **Z**. The index of that loop determines what letter goes in the `GLabel` that is part of the compound.

In addition to creating the `GCompound` for the key, you also have to add it to the `GWindow`. To do so, you have to know where it belongs on the screen. There isn't any magical way to figure out the location of each key; the only way to figure out where each key goes is to take out a ruler and measure the distance from the upper left corner of the image to the center of each key. But don't panic. We've already done that for you. Immediately after the definitions shown in Figure 3, `EnigmaConstants.js` includes a constant that specifies the *x* and *y* coordinates of the center of each key. The definition of that constant begins with the lines

```
const KEY_LOCATIONS = [
   { x: 140, y: 566 } /* A */,
   { x: 471, y: 640 } /* B */,
   { x: 319, y: 639 } /* C */,
   { x: 294, y: 567 } /* D */,
```

and ends some lines later with

```
   { x: 168, y: 639 } /* Y */,
   { x: 497, y: 496 } /* Z */
];
```

Figure 3. The constants used to define the appearance of a key

```
/* Constants that define the keys on the Enigma keyboard */

const KEY_RADIUS = 24;                  /* Outer radius of a key in pixels */
const KEY_BORDER = 3;                   /* Width of the key border         */
const KEY_BORDER_COLOR = "#CCCCCC";     /* Fill color of the key border    */
const KEY_BGCOLOR = "#666666";          /* Background color of the key     */
const KEY_UP_COLOR = "#CCCCCC";         /* Text color when the key is up   */
const KEY_DOWN_COLOR = "#CC3333";       /* Text color when the key is down */
const KEY_LABEL_DY = 10;                /* Offset from center to baseline  */
const KEY_FONT = "Helvetica Neue-Bold-28";
```

The value of the **KEY_LOCATIONS** constant is an array with 26 elements corresponding to the 26 letters. The element at index 0 corresponds to the letter **A**, the element at index 1 corresponds to the letter **B**, and so on. Each of the elements represents a point in space, as defined in section 8.2 of the text. For example, the key for **A** has its center at (140, 566) relative to the top left corner of the window. Given a character **ch**, the expression **KEY_LOCATIONS[ch.charCodeAt(0) - "A".charCodeAt(0)].x** produces the *x* coordinate of the key for **ch**; selecting the **.y** field produces the *y* coordinate.

Those definitions are all you need to get the keys displayed on the screen, and you should definitely get this part working before you move on to the next milestone. Of course, it may not be easy to tell whether your program is working because—at least when you've done everything right—the keys sit exactly on top of the identical image in the background. To make sure that things are working, you need to be a bit clever. One thing you can do is change the border color of the key defined by the **GCompound**. If that change shows up on the screen, you can check this milestone off your list.

Milestone #2: Detect "mousedown" and "mouseup" events in the keys

The version of the program you created at the end of the last milestone does not yet respond to any mouse actions. For this milestone, your job is to add the necessary action listeners to the **GCompound** for each key so that a **"mousedown"** event changes the color of the label on the key to **KEY_DOWN_COLOR**. Similarly, keys need to define a **"mouseup"** event that changes the color back to **KEY_UP_COLOR**.

This part of the assignment is in some ways just as simple as it sounds and will take no more than a few lines of code once you've figured out what to do. When you define a key, you need to assign functions (both of which take **enigma** as a parameter as described on page 4) to the **mousedownAction** and **mouseupAction** fields of the key. What takes some deeper thinking is figuring out how the callback function knows what key it's working with and how to get access to the **GLabel** to change the color. Getting this part of the solution to work depends on understanding closures and recognizing that the callback function has access to the variables of the function in which it was defined.

Milestone #3: Create the lamp panel

The area above the Enigma keyboard contains 26 lamps organized in the same pattern as the keyboard. The process of creating the lamps is similar to that of creating the keys and is again controlled by constants in the **EnigmaConstants.js** file, as shown in Figure 4. As with the keys, these constants are followed by a constant that shows the position of each lamp.

Figure 4. The constants used to define the appearance of a lamp

```
/* Constants that define the lamps above the Enigma keyboard */

const LAMP_RADIUS = 23;                /* Radius of a lamp in pixels      */
const LAMP_BORDER_COLOR = "#111111";   /* Line color of the lamp border   */
const LAMP_BGCOLOR = "#333333";        /* Background color of the lamp    */
const LAMP_OFF_COLOR = "#666666";      /* Text color when the lamp is off */
const LAMP_ON_COLOR = "#FFFF99";       /* Text color when the lamp is on  */
const LAMP_LABEL_DY = 9;               /* Offset from center to baseline  */
const LAMP_FONT = "Helvetica Neue-Bold-24";
```

For the most part, the code to display the lamps mirrors the code you wrote to display the keys. The are only a couple of differences that you need to take into account. First, the **GCompound** for a lamp is simpler to create because it has only two parts: a **GOval** for the lamp (outlined in **LAMP_BORDER_COLOR** and filled in **LAMP_BGCOLOR)**, and a **GLabel** for the letter on top of the lamp. Second, lighting a lamp is much easier to implement if you add a **label** field to the **GCompound** that takes you immediately to the label. Thus, if the **GCompound** is stored in the variable **lamp** and the **GLabel** is stored in the variable **label**, you can create this reference by writing

```
lamp.label = label;
```

JavaScript allows programmers to add new fields to an existing object, even if it is a library-defined object such as a **GCompound**. The variable **lamp** is still a **GCompound** and can still be added to a **GWindow**, but it also allows you to change its label color like this:

```
lamp.label.setColor(LAMP_ON_COLOR);
```

Milestone #4: Connect the keyboard and lamp panel

Unlike the keyboard, the lamp panel does not respond to mouse events that occur when you press or click the mouse on one of the lamps. In the fullness of time, you want pressing one of the keys[2] to light the lamp corresponding to the encrypted version of that character after the signal passes through all the encryption rotors. Before you do all that coding, however, it makes sense to see whether you can extend your program so that pressing one of the keys—in addition to changing the key color—also lights the corresponding lamp. Thus, if you press the **A** key, for example, you would like the label on the **A** lamp to appear to glow by changing the text color to the yellowish shade defined by the constant **LAMP_ON_COLOR.**

Once again, this task may initially seem simple. For Milestone #2, you implemented the **mousedownAction** for keys that already changes the key color. All that code has to do is change the color of the lamp as well. Changing the lamp color, however, is slightly more difficult than changing the key color. For keys, the **mousedownAction** callback function is defined within the code that creates the key and therefore has access to the label as a local variable. The code that creates each individual key, however, doesn't know anything about the lamps and therefore has to determine the JavaScript object for the lamp in a different way.

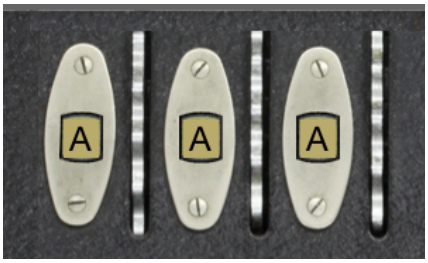
The solution to this problem lies in the fact that the **mousedownAction** function receives the data structure for the Enigma machine as a parameter. In order to light one of the lamps when you press a key, the JavaScript object stored in the **enigma** parameter must contain enough information to allow the callback function to find a lamp given the character that appears in its label. The simplest strategy is to include an array of all the lamps as one of the fields in **enigma**. The callback function could then select the element at the appropriate position in the alphabet to retrieve the lamp object. Once you have retrieved that object, you can use the code from the end of the previous section to change its label color.

---

[2] In this handout, we're using "pressing a key" as shorthand for "pressing the mouse button on top of one of the Enigma keys"; the physical keyboard on your computer is not involved.

Milestone #5: Add the rotors in their original positions

The encryption rotors for the machine are positioned underneath the top of the Enigma panel. The only part of a rotor you can see is the single letter that appears in the window at the top of the Enigma image. For example, in the diagram shown in Figure 1, you see the letters



at the top of the window. The letters visible through the windows are printed along the side of the rotor and mounted underneath the panel so that only one letter is visible at a time for each rotor. Whenever a rotor turns one notch, the next letter in the alphabet appears in the window. The act of moving a rotor into its next position is called ***advancing*** the rotor and is discussed in more detail in the description of Milestone #6.

For this milestone, your job is to implement the rotors in their initial position in which the **A** shows on each rotor. As with the keys and lamps, rotors have a visible component implemented as a `GCompound`. In this case, the `GCompound` includes a small `GRect` colored to match the piece of the rotor visible through the window and a `GLabel` that shows the letter. As in the earlier structures, the appearance of the `GRect` and `GLabel` are controlled by constants, which are shown in Figure 5, along with an array that gives the *x* and *y* coordinates of the three rotors from left to right. As noted in Handout #36, the rotor on the left is called the ***slow rotor,*** the one in the middle is the ***medium rotor,*** and the one on the right is the ***fast rotor.***

Although you have already had a little practice adding new fields to a `GCompound` when you created the keys and lamps, the rotor is more like an iceberg in the sense that most of it is invisible below the surface. In addition to the visible component that

Figure 5. The constants used to define the appearance of the rotor displays

```
/* Constants that control the display of the current rotor setting */

const ROTOR_BGCOLOR = "#BBAA77";       /* Background color for the rotor  */
const ROTOR_WIDTH = 24;                /* Width of the setting indicator  */
const ROTOR_HEIGHT = 26;               /* Height of the setting indicator */
const ROTOR_COLOR = "Black";           /* Text color of the rotor         */
const ROTOR_LABEL_DY = 9;              /* Offset from center to baseline  */
const ROTOR_FONT = "Helvetica Neue-24";

/* This array specifies the coordinates of each rotor display */

const ROTOR_LOCATIONS = [
   { x: 244, y: 95 },
   { x: 329, y: 95 },
   { x: 412, y: 95 }
];
```

appears in the display window, each of the rotors defines a letter-substitution cipher by specifying a permutation of the alphabet in the form of a 26-character string. The permutations for each of the three rotors—which in fact correspond to actual Enigma rotors—are supplied in the **EnigmaConstants.js** file in the form of the following array:

```
const ROTOR_PERMUTATIONS = [
  "EKMFLGDQVZNTOWYHXUSPAIBRCJ",
  "AJDKSIRUXBLHWTMCQGZNPYFVOE",
  "BDFHJLCPRTXVZNYEIWGAKMUSQO"
];
```

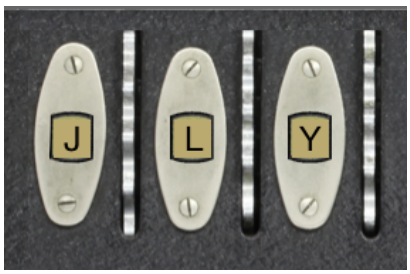
The permutation for the slow rotor, for example, is **"EKMFLGDQVZNTOWYHXUSPAIBRCJ"**, which corresponds to the following character mapping:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
E K M F L G D Q V Z N T O W Y H X U S P A I B R C J
```

At the moment, you don't have to implement the permutation; you will have a chance to do that in Milestone #7. All you need to do is store the permutation for each rotor as part of the JavaScript object that defines the rotor. For example, if you're creating the slow rotor, you have to store the string **"EKMFLGDQVZNTOWYHXUSPAIBRCJ"** as a new field in the **GCompound** you create to display that rotor. Storing that information in a field means that your code will have access to that permutation whenever it has a reference to the rotor object.

Milestone #6: Implementing click actions for the rotors

When the German Enigma operator wanted to send a message, the first step in the process was to set the position of the rotors to a particular three-letter setting for that day. The settings for each day were recorded in a codebook, which the British codebreakers were never able to obtain. For example, if the codebook indicated that the setting of the day were **JLY**, the operator would manually reposition the rotors so that they looked like this:



In the Enigma simulator, you enter the day setting by clicking on the rotors. Each mouse click advances the rotor by one position. To enter the setting **JLY**, for example, you would click 9 times on the slow rotor, 11 times on the medium rotor, and 24 times on the fast rotor.

Implementing the click operation to advance the rotor requires just a few changes. First, you have to add a **clickAction** function to the rotor object, which will be called when the click occurs. Ideally, the body of the **clickAction** function is a single line that

calls a function to advance the rotor on which the click occurred. We've called that function **`advanceRotor`** and will refer to it by that name in the subsequent discussion. The **`advanceRotor`** function takes the rotor object that should advance as an argument.

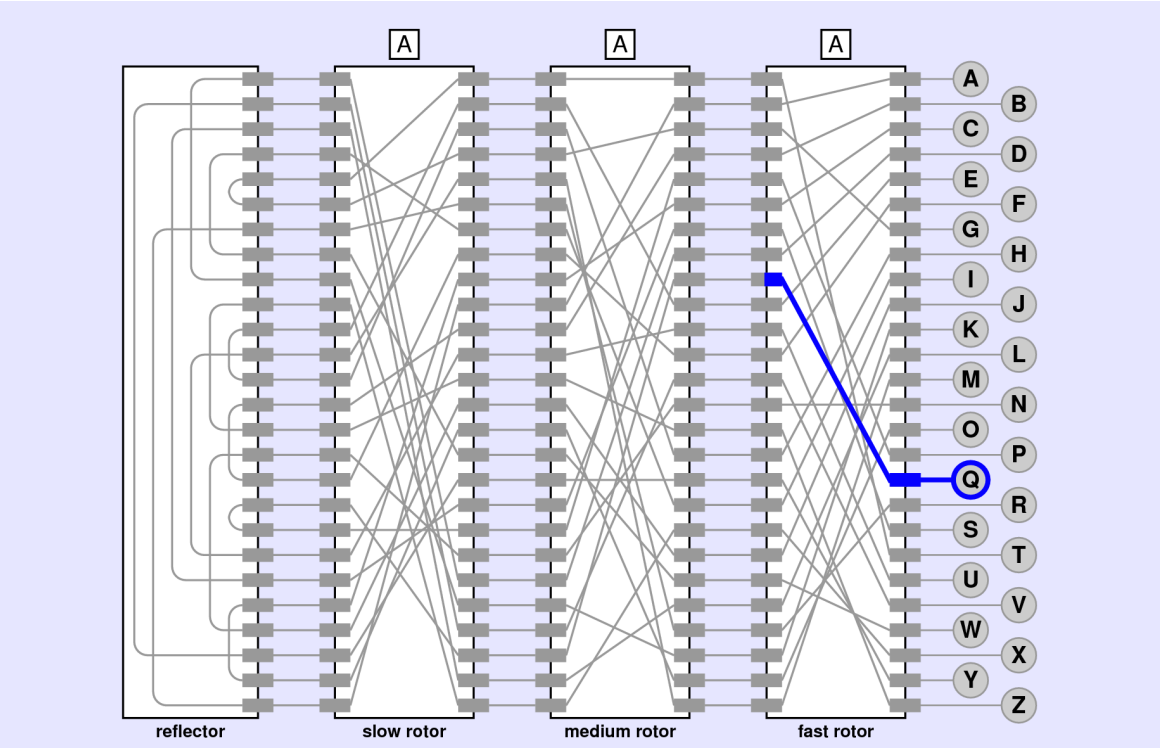
Implementing **`advanceRotor`** requires adding another data field to the rotor object. In addition to the permutation you added for Milestone #5, you also need to store the ***offset*** for the rotor, which is the number of positions the rotor has advanced. When the offset is 0 (as it is for all rotors in Milestone #5), the letter in the display is **A**. When the rotor advances one position, the offset changes to 1 and the letter in the display changes to **B**. As in a Caesar cipher, advancing a rotor is cyclical. If the offset is 25, which means that the display shows the letter **Z**, advancing the rotor changes the offset back to 0 and displays the letter **A**.

Milestone #7: Implement one stage in the encryption

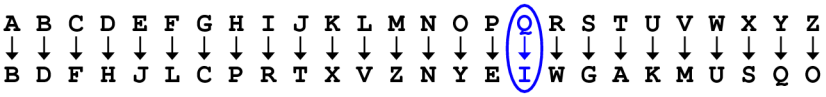
So far in this assignment, you've implemented lots of graphics and the underlying data structures, but haven't as yet done any encryption. Since Milestone #4, your program has responded to pressing the letter **Q** by lighting the lamp for the letter **Q**. Had the Germans used a machine that simple, the codebreakers could have all gone home.

The Enigma machine encrypts a letter by passing it through the rotors, each of which implements a simple letter-substitution cipher of the sort that Eric presented in class. Instead of trying to get the entire encryption path working at once, it makes much more sense to trace the current through just one step of the Enigma encryption. For example, suppose that the rotor setting is **AAA** and the Enigma operator presses the letter **Q**. Current flows from right to left across the fast rotor, as shown in Figure 6. The wiring

Figure 6. First step on the encryption path when the operator presses Q



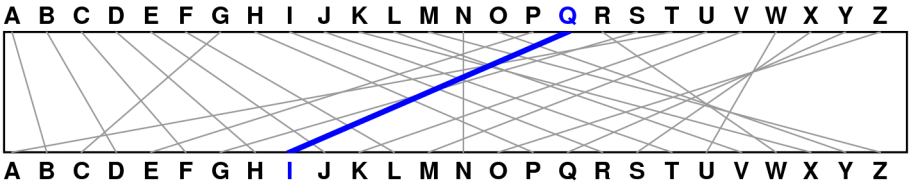
inside the fast rotor maps that current to the letter **I**, which you can determine immediately from the permutation for the fast rotor, which looks like this:



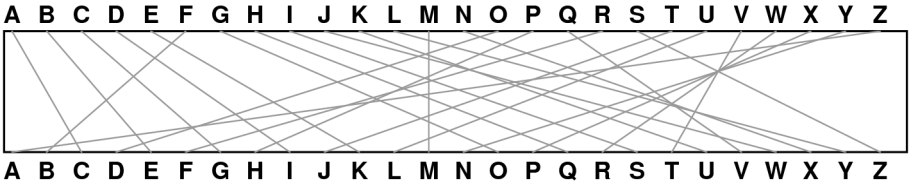
So far, so good. The process of translating a letter, however, is not quite so simple because the rotor may not be in its initial position. What happens if the offset is not 0?

One of the best ways to think about the Enigma encryption is to view it as a combination of a letter-substitution cipher and a Caesar cipher, because the offset rotates the permutation in a cyclical fashion similar to the process that a Caesar cipher uses. If the offset of the fast rotor were 1 instead of 0, the encryption would use the wiring for the next letter in the alphabet. Thus, instead of using the straightforward translation of **Q** to **I**, the program would need somehow to apply the transformation implied by the next position in the permutation, which shows that **R** translates to **W**.

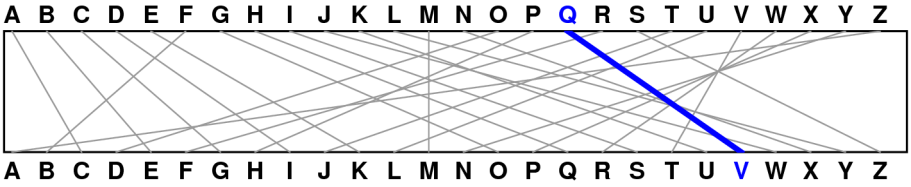
Focusing on the letters, however, is likely to get you confused. The problem is that the positions of the letters on each end of the rotor chain are fixed. After all, the lamps and keys don't move. What happens instead is that the rotor connections move underneath the letters. It is therefore more accurate to view the transformation implemented by the fast rotor in the following form:



When the rotor advances, the letters maintain their positions, but every wire rotates one position to the left with respect to the top and bottom connections. After advancing one notch, the rotor looks like this:



The change in the wiring is easiest to see in the straight line that connects **N** to **N** in the original state. After advancing, this wire connects **M** to **M**. If you press the **Q** key again in this new configuration, it shows up on the other side of the rotor as **V**:



What's happening in this example is that the translation after advancing the rotor is using the wire that connects **R** to **W** in the initial rotor position. After advancing, that wire connects **Q** and **V**, each of which appears one letter earlier in the alphabet. If the rotor has advanced *k* times, the translation will use the wire *k* steps ahead of the letter you're translating. Similarly, the letter in that position of the permutation string shows the letter *k* steps ahead of the letter you want. You therefore have to add the offset of the rotor to the index of the letter before calculating the permutation and, after doing so, subtract the offset from the index of the character you get. In each case, you need to remember that the process of addition and subtraction may wrap around the end of the alphabet. You therefore need to account for this possibility in the code in much the same way that the Caesar cipher does. The remainder operator will come in very handy here, but you need to keep in mind that the answer will be correct only if the indices never become negative.

The strategy expressed in the preceding paragraph can be translated into the following pseudocode function:

```
function applyPermutation(index, permutation, offset) {
   Compute the index of the letter after shifting it by the offset, wrapping around if necessary.
   Look up the character at that index in the permutation string.
   Return the index of the resulting character after subtracting the offset, wrapping if necessary.
}
```

If you implement this function and call it as part of your `"mousedown"` and `"mouseup"` event handlers, your simulation should implement this first stage. Pressing the **Q** key when the rotor setting is **AAA** should light up the lamp for the letter **I**. If you click on the fast rotor to advance the rotor setting to **AAB**, clicking **Q** again should light the lamp for the letter **V**.

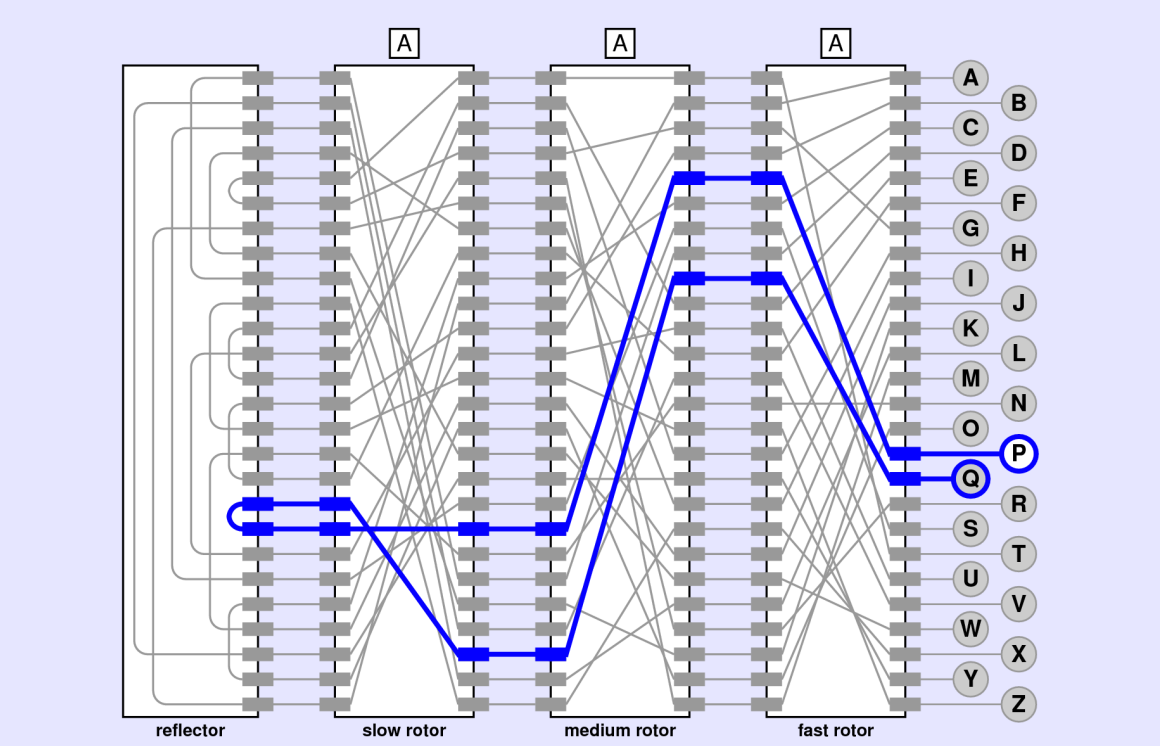
Milestone #8: Implement the full encryption path

Once you have completed Milestone #7 (and, in particular, once you have implemented a working version of `applyPermutation`), you're ready to tackle the complete Enigma encryption path. Pressing a key on the Enigma keyboard sends a current from right to left through the fast rotor, through the middle rotor, and through the slow rotor. From there, the current enters the reflector, which implements the following fixed permutation for which the offset is always 0:

```
const REFLECTOR_PERMUTATION = "IXUHFEZDAOMTKQJWNSRLCYPBVG";
```

When the current leaves the reflector, it makes its way backward from left to right, starting with the slow rotor, moving on to the medium rotor, and finally passing through the fast rotor to arrive at one of the lamps. The complete path that arises from pressing **Q** is illustrated in Figure 7 at the top of the next page. As you can see from the diagram, the current flows through the fast rotor from right to left to arrive at **I** (index 8), through the medium rotor to arrive at **X** (index 23), and through the slow rotor to arrive at **R** (index 17). It then makes a quick loop through the reflector and emerges at **S** (index 18). From there, the current makes its way backward through the slow rotor straight across to **S** (index 18), through the medium rotor to **E** (index 4) and finally landing at **P** (index 15).

Figure 7. Complete trace of the encryption path when the operator presses Q



If everything is working from your previous milestones, making your way through the rotors and the reflector should be straightforward. The challenge comes when you need to move backward through the rotors. The permutation strings show how the letters are inverted when you move through a rotor from right to left. When the signal is running from left to right, however, you can't use the permutation strings directly because the translation has to happen "backwards." What you need is the ***inverse*** of the original permutation.

The idea of inverting a key is most easily illustrated by example. Suppose, for example, that the encryption key is `"QWERTYUIOPASDFGHJKLZXCVBNM"`, which is the example Eric used in class. That key represents the following translation table:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
Q W E R T Y U I O P A S D F G H J K L Z X C V B N M
```

The translation table shows that **A** maps into **Q**, **B** maps into **W**, **C** maps into **E**, and so on. To turn the encryption process around, you have to read the translation table from bottom to top, looking to see what plaintext letter gives rise to each possible letter in the ciphertext. For example, if you look for the letter **A** in the ciphertext, you discover that the corresponding letter in the plaintext must have been **K**. Similarly, the only way to get a **B** in the ciphertext is to start with an **X** in the original message. The first two entries in the inverted translation table therefore look like this:

```
A B
↓ ↓
K X
```

If you continue this process by finding each letter of the alphabet on the bottom of the original translation table and then looking to see what letter appears on top, you will eventually complete the inverted table, as follows:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
K X V M C N O P H Q R S Z Y I J A D L E G W B U F T
```

The inverted key is simply the 26-letter string on the bottom row, which in this case is **"KXVMCNOPHQRSZYIJADLEGWBUFT"**.

To complete this milestone, you should implement the function **invertKey**, which takes a 26-letter string and returns its inverse. You can then call **invertKey** when you create the rotors so that each rotor contains both a right-to-left and a left-to-right version of the permutation. The offset applies to each of these permutations in exactly the same way.

Milestone #9: Implement the rotor advance on pressing a key

The final step in the creation of the Enigma simulator is to implement the feature that advances the fast rotor every time a key is pressed.[3] When the fast rotor has made a complete revolution, the medium rotor advances. Similarly, when the medium rotor makes a complete revolution, the slow rotor advances.

Given that you already have code that responds to **"mousedown"** events on the keys and a function to advance a rotor, the only new capability you need to implement is the "carry" from one rotor to the next. A useful technique to get this process working is to change the definition of **advanceRotor** so that it returns a Boolean: **false** in the usual case when no carry occurs, and **true** when the offset for that rotor wraps back to 0. The function that calls **advanceRotor** can check this result to determine whether it needs to propagate the carry to the next rotor.

When you complete this milestone, you're done!

Strategy and tactics

As with all assignments in CS 106J, the most important advice is to start early. You have 11 days to complete this assignment, which means that you can complete one milestone a day and still take the weekend off. Beyond that, several of the milestones require less than ten lines of new code. For those, you can almost certainly complete two or even three milestones in a day. What almost certainly won't work is if you start at 5:00P.M. on Thursday the 25th and try to finish all nine milestones in 24 hours. It is also particularly important for this assignment to complete each milestone before moving on to the next. Most of the later milestones depend on the earlier ones, and you need to know that the earlier code is working before you can debug the code you've added on top of the existing base.

---

[3] It is important to note that some sources (and, indeed, most of the movies) suggest that the fast rotor advances *after* the key is pressed. If you watch the Enigma machine work, however, it is clear that it is the force of the key press that advances the rotor. Thus, the fast rotor advances *before* the translation occurs.

The following tips will also probably help you do well on this assignment:

- *Try to get into the spirit of the history.* Although this project is an assignment in 2017, it may help to remember that the outcome of World War II once depended on people solving precisely this problem using tools that were far more primitive that the ones you have at your disposal. Computing sometimes matters a great deal, and there will probably be situations in your lifetimes when the consequences of solving some programming challenge will be just as important. The fate of the world may some day lie in your hands, just as it did for the cryptographers at Bletchley Park.
- *Draw lots of diagrams.* Understanding how the Enigma machine works is an exercise in visualizing how the machine works. A picture will be worth a thousand words here.
- *Debug your program by seeing what values appear in the variables.* When you are debugging a program, it is far more useful to figure out what your program *is* doing than trying to determine why it *isn't* doing what you want. Every part of this assignment works with strings, and you can get an enormous amount of information about what your program is doing by using `console.log` to display the value of the strings you're using at interesting points.
- *Check your answers against the demonstration programs.* The class web site includes a demo program that implements each milestone. Your code should generate the same results that the demo programs do.

Possible extensions

There are many things you can implement to make this assignment more challenging. Here are a couple of ideas:

- *Implement other features of the Enigma machine.* The German wartime Engima was actually more complicated than the model presented here. In particular, the wartime machines had a stock of five rotors of which the operators could use any three in any order. The Germans also added a plugboard (***Steckerbrett*** in German) that swapped pairs of letters before they were fed into the rotors and after they came out.
- *Simulate the actions of the Bombe decryption machine.* This assignment has you build a simulator for the German Enigma machine. Look at the extended documentation on Enigma decryption on the CS 106J web site and consider implementing some of the British decoding strategies.