Project 4: The Enigma Machine

Your task in this project 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 guide focuses on the necessary steps you need to accomplish to complete the assignment.

When you download the initial repository, the folder will contain three source files–`Enigma.py`, `EnigmaConstants.py`, and `EnigmaMachine.py`–along with an `images` folder. When you run the `Enigma.py` program as it stands, you should see the display shown below in Figure 1.

Figure 1: The output produced by the starting repository version of `Enigma.py`.

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

Take some time to look through the starting code in `Enigma.py`. You shouldn't need to modify this file unless you are implementing extensions, but it will be useful to read through it to make sure that you understand what it does. You need to make your modifications to the `EnigmaMachine` class and, as you move through the milestones, add the classes `EnigmaKey`, `EnigmaLamp`, and `EnigmaRotor`, each of which will be responsible for modeling one component of the machine. The code in `EnigmaMachine.py` defines the `EnigmaMachine` class, which has two purposes. First, it is responsible for creating the display, which consists

of creating new `GObject` instances of various kinds and then adding them to the window. Second, the `EnigmaMachine` class needs to keep track of whatever information about the state of the machine in necessary to run the simulation. It will, for example, maintain a list of all the keys and lamps, along with the three rotors that determine the encryption. In its current version, the only action the constructor takes is to read the file `EnigmaTopView.png` into a `GImage` and then add that image to the window. That is all that was needed to display the image shown in Figure 1.

The rest of this guide will describe the sequence of milestones that you should complete to create a working Enigma simulator. You can add the starting repository on Github by following the link here. You can see an online version of each milestone here to check your progress along the way.

Milestone 1: Create the keyboard

The first step on your way to implementing an interactive Enigma machine is to define a new `EnigmaKey` class that represents one of the keys on the keyboard. Like all the other classes you'll create from scratch, `EnigmaKey` is a subclass of `GCompound`, which means that it contains other graphical objects and that you can add it to the window. Your tasks for this milestone are to:

1. Define an `EnigmaKey` class that extends `GCompound`.
2. Implement the constructor for `EnigmaKey` so that it adds the necessary pieces.
3. Add code to `EnigmaMachine` to add 26 `EnigmaKey` objects to the `GWindow`

Referring back to Figure 1, a `EngimaKey` looks similar to the image below, which is a combination of a filled `GOval` with a gray border three pixels wide and a `GLabel` containing the letter `Q`.

The constants that define the appearance of a key are in `EnigmaConstants.py`, starting on line 68.

Your first step then for this milestone will be to create a new file called `EnigmaKey.py` and then add the necessary code to implement an initial version of the `EnigmaKey` class. All you need for now is the constructor, which takes as an argument a one-character string indicating the letter that appears on the key. Your code for the constructor must create the `GOval` and the `GLabel`, set their parameters so that the colors, fonts, and line widths match the values in the `EnigmaConstants.py` file, and then add those objects to the `GCompound`, which is stored in the `self` parameter.

Although you won't need to make use of this information until you implement Milestone 2, your implementation of the `EnigmaKey` constructor should store a reference to the `GLabel`

as an attribute of `self` so that it is possible for other methods in the class to change the color of the text. For example, when you press the mouse button on top of a key, you will eventually want to change the color of its `GLabel` to `KEY_DOWN_COLOR`, which is defined in the `EnigmaConstants.py` file as the hexadecimal string `"#CC3333"`. After you change the color of the label, the `Q` key will look like:

It is, of course, insufficient to just create one `EnigmaKey` object for the letter `Q`. What you need to do instead is to create 26 such objects, one for each letter of the alphabet. 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 is passed to the `EnigmaKey` constructor.

The task of creating the `EnigmaKey` objects is the responsibility of the `EnigmaMachine` constructor. After displaying the background image on the window (which is part of the starting code that you were given), you will need to execute a loop that creates each of the 26 keys and then adds it to the `GWindow`. To do so, you need to know where each key 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! That's already been done for you. Immediately after the definitions for the various key constants, `EnigmaConstants.py` includes a constant called `KEY_LOCATIONS` that specifies the x and y coordinates of the center of each key as a tuple containing the x and y coordinates. The value of the `KEY_LOCATIONS` constant is a list 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 on the window. So the key for `A` has its center at the point (140, 566).

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 have done everything correctly–the keys sit exactly on top of the identical image in the background. To make sure that things are working, you just want to be a bit clever. One easy option is that you could change the font color of the `GLabel` stored inside the `EnigmaKey` class. If that change shows up on your screen and all the keys are properly appearing in the correct location, you can go ahead and check this milestone off your list.

Milestone 2: Respond to mouse events in the keys

The version of the program you've created at the end of the last milestone does not yet respond to any sort of mouse actions. For this milestone, your job is to add the necessary callback functions to the `EnigmaKey` class so that a `"mousedown"` event changes the color of

the label to KEY_DOWN_COLOR. Similarly, you 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. Most of the work has already been done in the Enigma.py file. If you look back at the code in Enigma.py, you will see that the Enigma function defines three event listeners–one for each of the "mousedown", "mouseup" and "click" events–and then adds those listeners to the graphics window. The implementation of each of these listeners is almost identical, which means that if you understand one, you will immediately understand the others. The implementation of the mousedown_action looks like this:

```
def mousedown_action(e):
    gobj = gw.get_element_at(e.get_x(), e.get_y())
    if gobj is not None:
        if getattr(gobj, "mousedown_action", None) is not None:
            gobj.mousedown_action(enigma)
```

The mousedown_action function begins by using get_element_at to find the GObject that lives at the current mouse position. If there is such an object, the code then uses the built-in getattr function to see if that object defines a method called mousedown_action. If so, the code in that listener calls that method, passing in the value of enigma, which is the EnigmaMachine object that keeps track of the state of the Enigma machine. If there is no mousedown_action defined for the clicked object, the event is simply ignored.

The advantage of adding these listeners to the main program is that it is now extremely easy to make one of your graphical objects respond to a mouse event. All you need to do is define a method:

```
def mousedown_action(self, enigma):
```

in your desired object class. If that method is defined, it will be called whenever the mouse button is pressed inside that type of object.

Your job in Milestone 2 is therefore to add mousedown_action and mouseup_action methods to the EnigmaKey class. Assuming that you understand what you have to do and that you made sure the label of the key is accessible as an attribute to self, implementing this entire milestone should take exactly 4 lines of code. If you find yourself adding many more, you are probably on the wrong track.

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.py file, starting on line 110. As with the keys, these constants are followed by a constant that shows the position of each lamp. The code to display the lamps is almost exactly the same as the code you wrote to display the keys in Milestone 1, so you should be all set to go.

There are, however, a couple of extra methods that you need to implement for the `EnigmaLamp` class that arise from the fact that changes in the state of the lamp are triggered by events that occur elsewhere in the Enigma machine. An `EnigmaKey` changes color when you press the mouse button on top of it, which in turn triggers a call to the `mousedown_action` method. In an Enigma machine though, there is no physical action that corresponds to pressing the mouse button over a lamp, which means that the `EnigmaLamp` class won't need to define any callback functions.

What you need to do instead is include a getter and a setter method in the `EnigmaLamp` class that allows its clients to turn the lamp on and off. Thus, if the variable `lamp` contains a reference to an `EnigmaLamp` object, you should be able to turn the lamp on by calling

```
lamp.set_state(True)
```

and turn it off by calling

```
lamp.set_state(False)
```

The getter method allows you to determine whether the lamp is on by calling

```
lamp.get_state()
```

Milestone 4: Connecting 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 end, you want pressing one of the keys 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`.

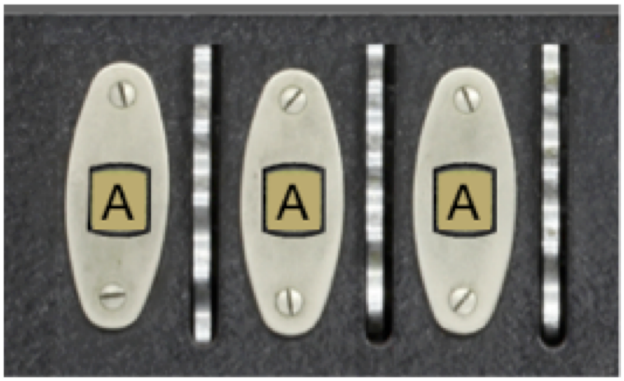
Once again, this task may initially seem simple. For Milestone 2, you implemented the `mousedown_action` in the `EnigmaKey` class so that it changes the key color. Changing the lamp color, however, is slightly more difficult. For keys, the `mousedown_action` method is defined within the code that creates the key and therefore has access to the label as an attribute of the object. The code that creates each individual key, however, does not know *anything* about the lamps. Moreover, if you remain true to the principles of encapsulation and information hiding, the keys *should not* know anything about the lamps! Working with lamps is the responsibility of the `EnigmaMachine` class.

The best way to maintain the abstraction barrier is to have the `mousedown_action` and `mouseup_action` methods pass along the fact that the user pressed or released that key on the keyboard to the `EnigmaMachine` object by calling a method on the `enigma` parameter that was passed in. If, for example, the `EnigmaMachine` class contains `key_pressed` and `key_released` methods, the callback functions in `EnigmaKey` could call those methods and leave the rest of the implementation up to the `EnigmaMachine` class, which is where it belongs.

The code for those two methods can then call `set_state` on the appropriate `EnigmaLamp` object to achieve the desired effect.

Milestone 5: Adding the rotors to original positions

The encryption rotors for the machine are positioned underneath the top of the Enigma panel. The only part of the rotor that 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. When 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 in `EnigmaConstants.py` starting at line 42. A constant called `ROTOR_LOCATIONS` appears shortly thereafter which gives the x and y coordinates of the three rotors, ordered from left to right. As discussed in lecture, 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 attributes to a `GCompound` subclass when you wrote the `EnigmaKey` and `EnigmaLamp` classes, the `EnigmaRotor` class is more like an iceberg in the sense that most of it is invisible beneath the surface. In addition to the visible component that 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 rotors in the real Enigma machine–are supplied in the `EnigmaConstants.py` file in the form of the following array:

```
ROTOR_PERMUTATIONS = [
  "EKMFLGDQVZNTOWYHXUSPAIBRCJ",   # Permutation for slow rotor
  "AJDKSIRUXBLHWTMCQGZNPYFVOE",   # Permutation for medium rotor
  "BDFHJLCPRTXVZNYEIWGAKMUSQO"    # Permutation for fast rotor
]
```

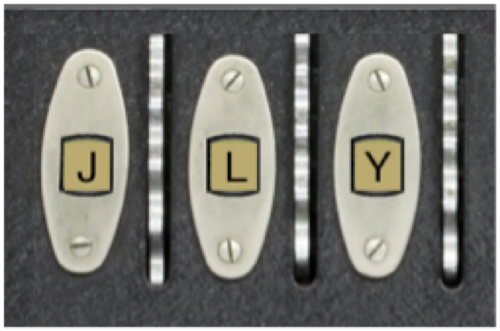
The permutation for the slow rotor, for example, is `"EKMFLGDQVZNTOWYHXUSPAIBRCJ"`, which corresponds to the following character mapping:

```
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 do not need to implement the permutation; you will have a chance to do that in Milestone 7. All you need to do here is store the permutation for each rotor as part of the Python object that defines the rotor. Fore example, if you are creating the slow rotor, you have to store the string `"EKMFLGDQVZNTOWYHXUSPAIBRCJ"` as a new attribute in the `EnigmaRotor` object.

Milestone 6: Implement 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 the day. The settings for each day were recorded in a codebook, which the British were never able to obtain. For example, if the codebook indicated that the setting of the day was `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 `click_action` method to the `EnigmaRotor` class, which will be called when the click occurs. Ideally, the body of the `click_action` method is a single line that calls a method to advance the rotor on which the click occurred. I have called that method `advance` and will refer to it by that name in the subsequent discussion.

Implementing `advance` requires adding another attribute 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 in `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 of encryption

So far in this assignment, you've implemented lots of graphics and the underlying classes, 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 all simply have 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 I showed in class. Instead of trying to get the entire encryption 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 below in Figure 2. The wiring inside the fast rotor maps the current

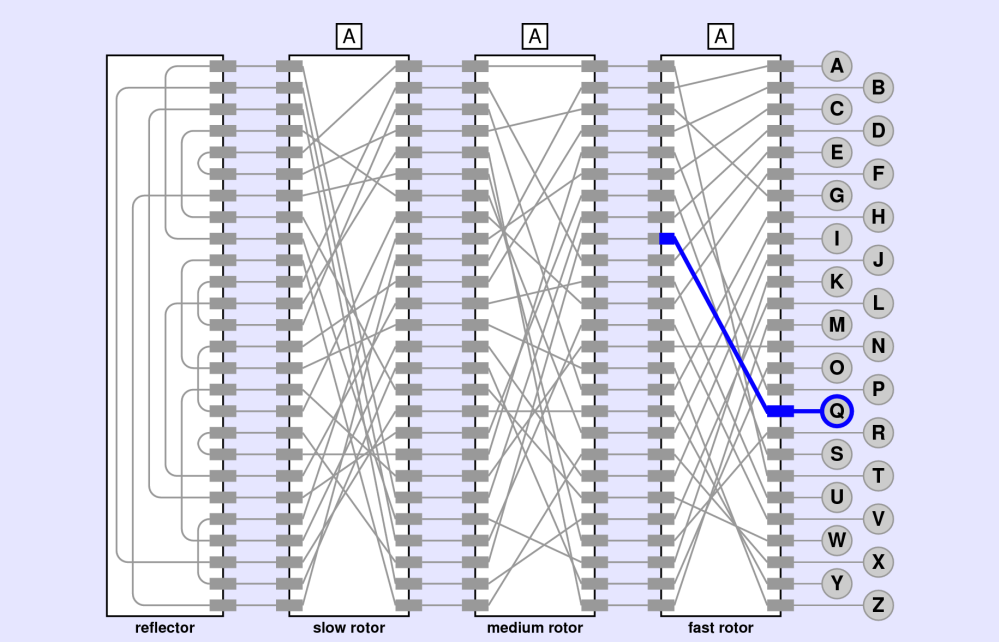
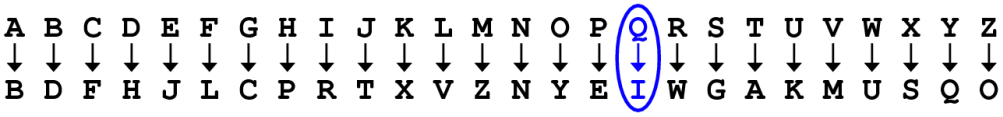


Figure 2: The first step of the encryption path when the operator presses `Q`.

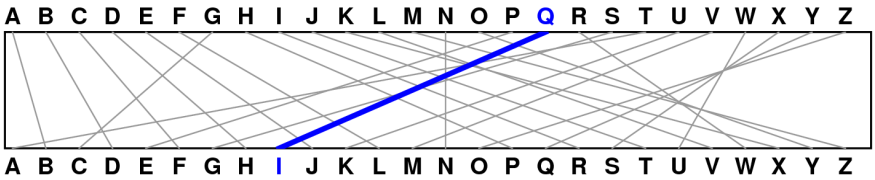
to the letter `I`, which you can determine immediately from the permutation for the fast rotor, which looks like:



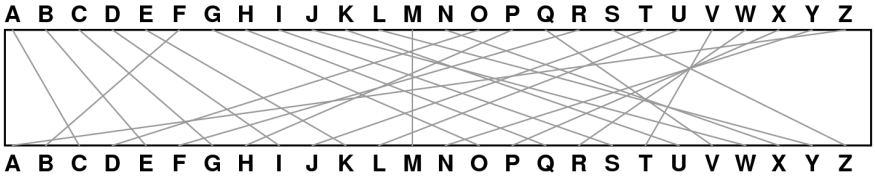
So far, so good. The process of translating the letter, however, is not quite so simple because the rotor may not be in its initial position. What happens if the offset is not 0?

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`.

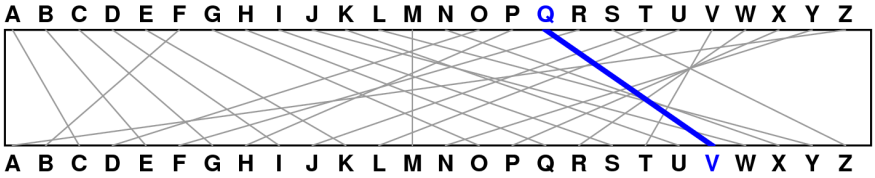
Focusing on the letters though is likely to get you confused. The problem is that the positions of the letters on each end of the rotor chain are fixed. After all, the lamps and the keys do not move. What happens instead is that the rotor connections move underneath the letters. It is therefore more 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 is 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 that you are translating. Similarly, the letter in that position of the permutation string shows the letter k steps ahead of the letter that 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 that 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.

The strategy expressed in the preceding paragraph can be translated into the following pseudocode method, which is easiest to implement as a top-level (not nested inside anything) function in the `EnigmaRotor.py` module:

```
def apply_permutation(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 new character after subtracting the
        offset, wrapping if necessary
```

If you implement this function and call it in the `key_pressed` and `key_released` methods in `EnigmaMachine` , 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 `apply_permutation`), you are 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 medium 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:

```
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 3. As you can see from the diagram, the current flows through the fast rotors 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

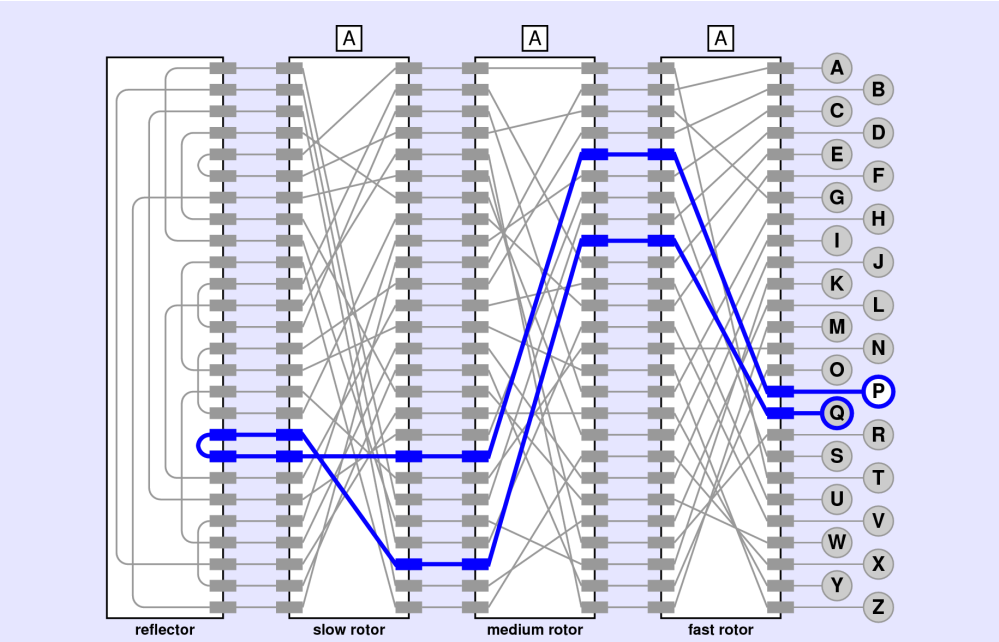


Figure 3: A complete trace of the encryption path when the operator pressed `Q`.

rotor to `E` (index 4) and finally landing at `P` (index 15). If everything is working from your previous milestones, making your way through the rotors and the reflector should not be too difficult. The challenge comes when you need to move backward through the rotors. The permutation strings as given show how the letters are transformed when you move through a rotor from right to left. When the signal is running from left to right, however, you can not 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 an example. Suppose that the encryption key is `"QWERTYUIOPASDFGHJKLZXCVBNM"`. That key represents the following encryption 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 to `Q`, `B` maps to `W`, `C` maps to `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 (bottom row), you discover that the corresponding letter in the plaintext (top row) 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 the 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 need to write a function `invert_key` (it can be a top-level function), which will take a initial permutation string as input and return the inverted key as a string. Then, when you create the rotors, you should call `invert_key` so that each rotor contains an attribute for *both* a right-to-left and a left-to-right version of the permutation. The offset applies to these permutations in the exact same way, so you don't need to worry about anything extra there. Then when you pass the signal through the rotor from left to right, you can just use that inverted permutation.

Milestone 9: Advance the rotor on pressing a key

The final step in the creating of the Enigma simulator is to implement the feature that advances the fast rotor every time a key is pressed. Although some sources (and most of the movies) suggest that the fast rotor advances *after* the key is pressed, watching the Enigma machine work makes it clear that it is the force of the key press that advances the rotor. Thus the fast rotor advances *before* the translation occurs. 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 `advance` 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 `advance` can check this result to determine whether it needs to propagate the carry to the next rotor.

When you complete this milestone, you are done!!

Strategy and Tactics

As with all projects in this class, the most important advice is to start early! You have enough time with this project that you could complete one milestone per day and still finish early. Beyond that, several of the milestones require less that ten lines of new code. For those, you can almost certainly complete two or even three milestones in a day. What almost certainly will not work is if you start the day before it is due and try to finish all nine

milestones in a single day. It is also important 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 have added on top of the existing base.

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

- *Try to get into the spirit of the history.* Although this project is an assignment in 2020, it may help to remember that the outcome of World War II once depended on the people solving this problem using tools that were far more primitive than the ones you have at your disposal.
- *Draw lots of diagrams.* Understanding how the Enigma machine works is an exercise in visualization. A picture is often worth a thousand words here, so draw them!
- *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 *is not* 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 `print` to display the value of the strings you are using at interesting points.
- *Check your answers against the demonstration programs.* I linked you above to a website which includes online demo programs for 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 few ideas:

- *Implement other features of the Enigma machine.* The German wartime Enigma was 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 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. Check out the extended documentation on Enigma decryption and consider implementing some of the British decoding strategies.