UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2012

Project #0: A Simple Enigma

Due: Thursday, 20 September 2012 at midnight

1 Introduction

This initial programming assignment is intended as an extended finger exercise, a mini-project rather than a full-scale programming project. True, there is quite a bit of background reading, but the necessary program is not (or rather need not be) terribly big. The intent is to give you a chance to get familiar with Java and the various tools used in the course.

We will be grading *solely* on whether you manage to get your program to work (according to our tests) and to hand in the assigned pieces. There is a slight stylistic component: the submission and grading machinery require that your program pass a mechanized style check (`style61b`), which mainly checks for formatting and the presence of comments in the proper places. See `http://inst.eecs.berkeley.edu/~cs61b/fa12/labs/style61b.txt` for a description of the style it enforces and how to run it yourself.

To obtain the skeleton files (and set up an initial entry for your project in the repository), you can use the command

```
$ hw init proj0
```

which creates a working directory `proj0`. You will also find these files in `~cs61b/code/proj0`.

2 Background

You've probably heard of the Enigma machines that Germany used during World War II to encrypt its military communications. This project involves building a simulator for a very simplified version of this machine[1]. Your program will take descriptions of possible initial

[1]There were actually a number of varieties of the machine, each implementing its own encryption algorithm. Ours is probably closest to M3, used by the German army and navy. We have left off the plugboard (*Steckerbrett,*) which added an additional configurable permutation to the encryption, increasing the number of possible configurations by a factor of roughly 5.5×10^{20}. We've also omitted the configurability of the alphabet ring (*Ringstellung*) on each rotor, which allowed changing the points at which the rotor sequenced.

configurations of the machine and messages to encode or decode (the Enigma algorithms were *reciprocal,* meaning that the encryption is its own inverse operation.)

The Enigmas effect a *substitution cipher,* on the letters of a message. That is, at any given time, the machine performs a permutation—a one-to-one mapping—of the alphabet onto itself. The alphabet consists solely of the 26 letters in one case (there were various conventions for spaces and punctuation).

Plain substitution ciphers are easy to break (you've probably seen puzzles in newspapers that consist of breaking such ciphers). The Enigma, however, implements a *progressive* substitution, different for each subsequent letter of the message. This made decryption considerably more difficult.

The device consists of a simple mechanical system of interchangeable *rotors* (*Walzen*) that sit side-by-side on a shaft and make electrical contact with each other. In our simulator, there will be a total of 8 rotors (labeled with roman numerals I–VIII,) of which three will be used in any given configuration, plus 2 special reflectors (labeled 'B' and 'C',) of which one will be used in any given configuration. Each rotor has 26 contacts on both sides, which are wired together internally so as to effect a permutation of signals coming in from one side onto the contacts on the other (and the inverse permutation when going in the reverse direction). The reflectors (*Umkehrwalzen*) are special "rotors" placed on the far left side that don't rotate during translation and have all of their contacts on one side. They simply connect half of their contacts to the other half. Figure 1 shows the permutations implemented by all the rotors and reflectors.

A signal starting from the right in one of the 26 possible positions will flow through wires in three rotors, "bounce" off the reflector, and then come back through the same rotors by a different route, always ending up permuted to a letter position different from where it started[2].

Each rotor and each reflector implements a different permutation, and the overall effect depends on their configuration: which rotors and reflector are used, what order they are placed in the machine, and which rotational position they are initially set to[3]. This configuration is the secret key used to encrypt or decrypt a message. On our simple machine, there are thus 307087872 possible configurations[4].

Before a letter of a message is translated, a spring-loaded pawl (lever), one for each of the three rotating rotors, tries to engage a ratchet on the right side of the rotor and to rotate that rotor by one position (so that signals that originally entered or exited the 'A' position on one side of the rotor will now enter or exit through the 'B' position, etc.), thus changing the permutation performed by the rotor. The lever on the right rotor always succeeds, so that the right (or "fast") rotor rotates one position before each character. The pawl pushing the

[2]A significant crytographic weakness, as it turned out. It doesn't really do a would-be code-breaker any good to know that some letters in an encrypted message *might* be the same as the those in the plaintext if he doesn't know which ones. But it does a great deal of good to be able to eliminate a possible decryption because one position in the encrypted text would have the same letter as the plaintext.

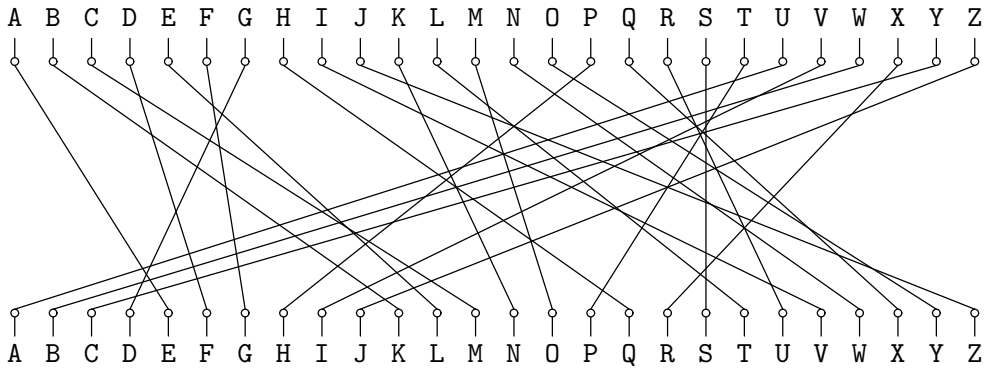
[3]Our machine differs from the M3 in that we allow the reflector to be set. It seems that the reflector in most of the real machines allowed but one or two positions, as do most simulation applets.

[4]Figuring 2 possible reflectors, $\begin{pmatrix} 8 \\ 3 \end{pmatrix} = 56$ choices of rotors, $3! = 6$ possible orders for those rotors, and 26^4 possible initial rotational settings.

middle and left rotors, however, rests over both the ratchet on its respective rotor and on a ring on the left side of the rotor to its right. This ring holds the pawl away from its ratchet, preventing its wheel from moving, except when the pawl is positioned over a notch in the ring. Each rotor has at least one such notch (two in the case of three of the possible rotors). A "feature" of the design[5] is that when a pawl is in a notch, it also moves the notch, so that the rotor on its right also moves. This means that the middle rotor experiences "double stepping"—if it advances on one step so that its notch moves under the pawl to its left, then on the next step, when the left rotor advances, it also advances the middle rotor (even though its own pawl has moved off the notch to its right on the previous move). This peculiarity only affects the middle rotor: the rightmost rotor always advances anyway, and the reflector (to the left of the left rotor) never moves.

The contacts on the rightmost rotor's right side connect with stationary input and output contacts, which run to keys that, when pressed, direct current to the contact from a battery or, when not pressed, direct current back from the contact to a light bulb indicating a letter of the alphabet. Since a letter never encrypts or decrypts to itself, the to and from directions never conflict.

The operator can set the initial positions of the rotors using finger wheels on each rotor. Each rotor has an alphabet running around its perimeter and visible to the operator through the top of the machine. The letter that is showing at any given time indicates the position of that rotor. Consider, for example, the rotor labeled 'I', which gives the following permutation, when the rotor is in its 'A' position:

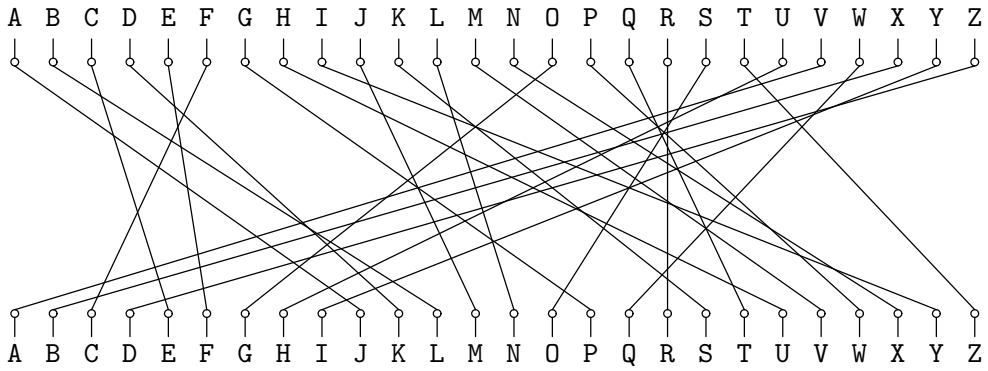


When this rotor is rotated by one position into its 'B' position, the permutation becomes:

[5]It was corrected in other versions of the Enigma, since it reduced the period of the cipher.

Rotor	Permutation (as cycles)	Notch
Rotor I	(AELTPHQXRU) (BKNW) (CMOY) (DFG) (IV) (JZ) (S)	Q
Rotor II	(FIXVYOMW) (CDKLHUP) (ESZ) (BJ) (GR) (NT) (A) (Q)	E
Rotor III	(ABDHPEJT) (CFLVMZOYQIRWUKXSG) (N)	V
Rotor IV	(AEPLIYWCOXMRFZBSTGJQNH) (DV) (KU)	J
Rotor V	(AVOLDRWFIUQ)(BZKSMNHYC) (EGTJPX)	Z
Rotor VI	(AJQDVLEOZWIYTS) (CGMNHFUX) (BPRK)	Z and M
Rotor VII	(ANOUPFRIMBZTLWKSVEGCJYDHXQ)	Z and M
Rotor VIII	(AFLSETWUNDHOZVICQ) (BKJ) (GXY) (MPR)	Z and M
Reflector B	(AY) (BR) (CU) (DH) (EQ) (FS) (GL) (IP) (JX) (KN) (MO) (TZ) (VW)	
Reflector C	(AF) (BV) (CP) (DJ) (EI) (GO) (HY) (KR) (LZ) (MX) (NW) (TQ) (SU)	

Figure 1: The rotors and their mappings. The notation here is *cycle representation.* For example, "(CMOY)" means " 'C' maps to 'M'; 'M' maps to 'O'; 'O' maps to 'Y', and 'Y' maps to 'C'. A singleton such as "(S)" means that 'S' maps to itself. The 'Notch' column indicates the position of a rotor at the point where its notch allows the rotor to its left to advance. Thus, when Rotor I is at 'Q' and the machine advances, the rotor to the left of Rotor I will advance. Source: Tony Sale's pages at `http://www.codesandciphers.org.uk/enigma/rotorspec.htm`



so that whereas 'B' converts to 'K' and 'D' converts to 'F' in the original ('A') position, 'A' converts to 'J' and 'C' converts to 'E' in the shifted ('B') position.

3 Input and Output

To run your program, you can use the command

```
java enigma.Main
```

This takes input from the terminal and produces output on the terminal, and so is not recommended for extended examples. For that, do as is done in the makefile for `gmake check`:

```
java enigma.Main < INPUT-FILE   # Output to terminal
```

or

```
java enigma.Main < INPUT-FILE   > OUTPUT-FILE
```

The input to your program will consist of a sequence of messages to decode, each preceded by a line giving the initial configuration. A configuration line looks like this:

```
* B III IV I AXLE
```

This particular example means that the rotors used are reflector B, and rotors III, IV, and I, with rotor I in the rightmost, or fast, slot. The first rotor is always the reflector, and the each of the remaining is one of rotors I–VIII. A rotor may not be repeated. The last four-letter word gives the initial positions of the four rotors, in the same order as they were specified.

After each configuration line comes a message on any number of lines. Each line of a message consists only of letters and blanks. The program should ignore the blanks and convert all letters to upper case. The end of message is indicated either by the end of the input or by a new configuration line (distinguished by its leading asterisk). The machine is not reset between lines, but continues stepping from where it left off on the previous message line.

Print the translation for each message line in groups of five upper-case letters, separated by blanks (the last group may have fewer characters, depending on the message length).

Because the Enigma is a reciprocal cipher, a given translation may either be a decryption or encryption; you don't have to worry about which, since the process is the same in any case.

Here is an example that shows an encryption followed by a decryption of the encrpyted message.

Input	Output

`* B III IV I AXLE`
`FROM his shoulder Hiawatha`
`Took the camera of rosewood`
`Made of sliding folding rosewood`
`Neatly put it all together`
`In its case it lay compactly`
`Folded into nearly nothing`
`But he opened out the hinges`
`Pushed and pulled the joints`
`  and hinges`
`Till it looked all squares`
`  and oblongs`
`Like a complicated figure`
`In the Second Book of Euclid`
 	
`* B III IV I AXLE`
`HYIHL BKOML IUYDC MPPSF SZW`
`SQCNJ EXNUO JYRZE KTCNB DGU`
`FLIIE GEPGR SJUJT CALGX SNCTM KUF`
`WMFCK WIPRY SODJC VCFYQ LV`
`QLMBY UQRIR XEPOV EUHFI RIF`
`KCGVS FPBGP KDRFY RTVMW GFU`
`NMXEH FHVPQ IDOAC GUIWG TNM`
`KVCKC FDZIO PYEVX NTBXY AHAO`
`BMQOP GTZX`
`VXQXO LEDRW YCMMW AONVU KQ`
`OUFAS RHACK`
`KXOMZ TDALH UNVXK PXBHA VQ`
`XVXEP UNUXT XYNIF FMDYJ VKH`
	`HYIHL BKOML IUYDC MPPSF SZW`
	`SQCNJ EXNUO JYRZE KTCNB DGU`
	`FLIIE GEPGR SJUJT CALGX SNCTM KUF`
	`WMFCK WIPRY SODJC VCFYQ LV`
	`QLMBY UQRIR XEPOV EUHFI RIF`
	`KCGVS FPBGP KDRFY RTVMW GFU`
	`NMXEH FHVPQ IDOAC GUIWG TNM`
	`KVCKC FDZIO PYEVX NTBXY AHAO`
	`BMQOP GTZX`
	`VXQXO LEDRW YCMMW AONVU KQ`
	`OUFAS RHACK`
	`KXOMZ TDALH UNVXK PXBHA VQ`
	`XVXEP UNUXT XYNIF FMDYJ VKH`
	
	`FROMH ISSHO ULDER HIAWA THA`
	`TOOKT HECAM ERAOF ROSEW OOD`
	`MADEO FSLID INGFO LDING ROSEW OOD`
	`NEATL YPUTI TALLT OGETH ER`
	`INITS CASEI TLAYC OMPAC TLY`
	`FOLDE DINTO NEARL YNOTH ING`
	`BUTHE OPENE DOUTT HEHIN GES`
	`PUSHE DANDP ULLED THEJO INTS`
	`ANDHI NGES`
	`TILLI TLOOK EDALL SQUAR ES`
	`ANDOB LONGS`
	`LIKEA COMPL ICATE DFIGU RE`
	`INTHE SECON DBOOK OFEUC LID`

4 Handling Errors

You can see a number of opportunities for input errors:

- The input might not start with a configuration.
- The configuration line can contain the wrong number of arguments.
- The rotors might be misnamed.
- A rotor might be repeated in the configuration.
- The first rotor might not be a reflector.

- The initial positions string might be the wrong length or contain non-alphabetic characters.
- The message might contain non-alphabetic characters.

A significant amount of a program will typically be devoted to detecting such errors, and taking corrective action. In our case, the only corrective action needed is in how you exit the program: use the Java call

```
System.exit(1);
```

(a normal exit is simply to return from the main program or to call `System.out(0)`.) In the case of an error, your output does not matter. Your program should *not* terminate as the result of an uncaught exception. The test script provided in the makefile checks that the program does not terminate with an exception.

5 What to Turn In

Use the command

```
$ hw submit proj0
```

(leave off the `proj0` if you are in the working directory already.) You will need to turn in all your Java files, test files, and make files. You may change *any* of the code we've provided, as long as the resulting program works according to the specifications here.

Your finished programs should be workmanlike, and of course, we will enforce the mechanical style standards. Make sure all methods are adequately commented—meaning that after reading the name, parameters, and comment on a method, you don't need to look at the code to figure out what a call will do. Don't leave debugging print statements lying around. In fact, don't use them; learn to use the debugger (either `gjdb` or that of `Eclipse` or your favorite Java-system vendor).

6 Advice

First, get started immediately, of course. Don't just jump in and code, though. Make sure you understand the specifications (which have their subtlties,) and plan out how you're going to meet them. Figure out how to break this problem down into small pieces, and how to implement and test them one piece at a time. Know in detail how you're going to do something before writing a line of Java code for it. In particular, take some time to understand how the rotors work, and especially how the rotor position modifies the permutation.

DBC: Don't allow things to remain mysterious to you, or they'll surely bite you at some point. You don't have to use *any* of the skeleton code we've provided as long as the command `java enigma.Main` works as specified. However, if you find yourself throwing something away because you don't understand it, *STOP!* You're allowing yourself to be mystified by something that is intended to be simple.

There is a fair amount of string-hacking involved. The Java library can help you. Look at the documentation of `String`, `Character`, and perhaps `java.util.regex.Pattern` in the on-line Java library documentation. We particularly invite you to consider

From `String`
- `charAt`
- `replaceAll`
- `split`
- `startsWith`
- `substring`
- `toUpper`
- `trim`

From `Character`
- `isLetter`
- `isWhitespace`

A useful property of characters is that the type `char` is actually an integer type, and that if `c` is an upper-case letter, then `c - 'A'` is the position of `c` in the alphabet ('A' is 0, 'B' is 1, etc.). Contrariwise, if `p` is the position of an upper-case letter in the alphabet, then `(char) (p + 'A')` is the corresponding character.

Be creatively lazy. For example, getting the rotor wiring right looks awfully tedious, so you should think about how you might cut-paste-and-edit the tables we provided into Java code that initializes the rotors. If you find yourself writing:

```
if (rotor.equals("I") {
    if (c == 'A')
        e = 'E';
    else if (c == 'B')
        e = 'K';
    ...
}
```

or something equally hideous, *STOP!* you are doing something wrong!

You can write acceptance tests of the overall system *before* you write a line of code. The commands in the makefile for `gmake check` allow you to create input files named `test/`*NAME*`.inp` and have them automatically run through your program and checked against standard output files named `test/`*NAME*`.out`. As you develop your system, use this directory to accumulate tests. Be particularly careful to include tests that at one point caused your program to fail (*regression tests*) so that you can be sure you don't backslide when you make further changes. There's no reason you can't have tests that you fail, by the way; they'll serve to remind you of things you still need to do.

Use `gmake style`. Your program will have to pass its tests eventually, and you don't want to have your submissions refused at the last minute because you have scads of style errors to fix. Also, we've put in '`//`' comments for you to replace to remind you of what needs to be done (you can use this trick, too, of course); `gmake style` will find these and make sure you haven't left something undone.

Use the repository. Frequently commit your work so that you'll never have to reconstruct too much if your files somehow vanish mysteriously.

Above all, it is always fair to ask for help and advice. We don't *ever* want to hear about how you've been beating your head against the wall over some problem for hours. If you can't make progress, don't waste your time guessing or bleeding: ask. If nobody's available to ask, do something else (or get some sleep).