P07 - A riddle, wrapped in a mystery, inside an Enigma

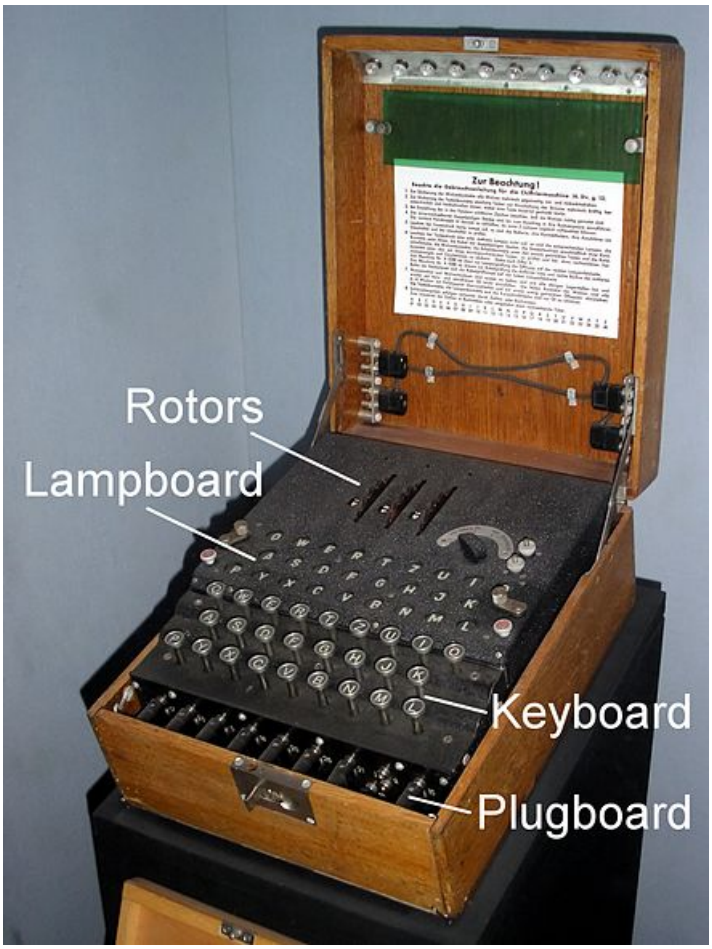
Due Tuesday, March 18 at 8 a.m.

CSE 1325 - Homework #7 - Rev 0

Assignment Background

> NOTE: With the conclusion of our Abuta project, we are back to the standard full_credit / bonus / extreme_bonus format for this our final *Java* homework assignment. Don't forget to create those subdirectories! You will *also* need a Tesults.txt file in the P07 directory (see below).

Running multiple threads in a single program is increasingly necessary to take full advantage of the multi-core hardware common in today's market. Let's put those cores to work, and explore the classic World War II encryption machine called Enigma!



Enigma Operation (optional)

You do NOT need to read this section to solve the assignment.

Germany used an encryption machine with 3 rotors (chosen from 8 rotors in the box), each of which could be started in one of 26 positions, and each with an associated ring that could also be started in one of 26 positions. They also had a plugboard to further complicate things, but we'll ignore that.

As each letter was typed on the keyboard, the associated (encrypted) letter lit up on the lampboard and the rotors then incremented like a 3-digit base-26 number. This encrypted each letter with a unique key. Enigma was a "reflective" device, meaning to decode a message you just encrypt the encrypted text from the exact same starting position.

Code Overview (optional)

The `enigma` package provided at cse1325-prof/P07/baseline is a (rather C-like) emulation of the WWII encryption device called Enigma. **You do NOT need to even LOOK at any code in package enigma!**

The `turing` package (named after Alan Turing, one of the heroes of Bletchley Park in England who continued the work begun in Poland to break Enigma) contains the encryption-breaking code.

You will ONLY modify file P07/full_credit/turing/BreakEnigmaFile.java for this entire assignment.

Full Credit

Copy the code from cse1325-prof/P07/baseline to your own cse1325/P07/full_credit directory and build it. Use a timer such as `time` in bash (or the equivalent in your chosen shell) to measure the runtime of `java turing/BreakEnigmaFile 10` which breaks 10 strings (it should take only a few seconds).

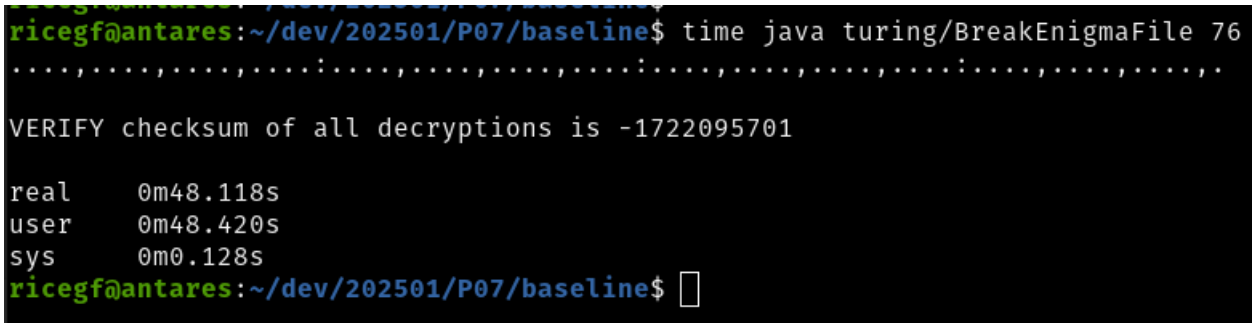
```
ricegf@antares:~/dev/202501/P07/baseline$ lt
Permissions Size User   Date Modified Name
drwxrwxr-x     - ricegf 03-05 08:55   .
.rw-rw-r--  1.2k ricegf 03-05 08:04   ├── build.xml
drwxrwxr-x     - ricegf 03-05 09:27   ├── enigma
.rw-rw-r--  3.4k ricegf 03-05 08:04   │   ├── Enigma.java
.rw-rw-r--  2.1k ricegf 03-05 08:04   │   ├── Plugboard.java
.rw-rw-r--   917 ricegf 03-05 08:04   │   ├── Reflector.java
.rw-rw-r--  3.8k ricegf 03-05 08:04   │   ├── Rotor.java
.rw-rw-r--  3.5k ricegf 03-05 08:04   │   └── Setting.java
.rw-rw-r--   15k ricegf 03-05 08:04   ├── input.txt
drwxrwxr-x     - ricegf 03-05 09:27   └── turing
.rw-rw-r--  7.6k ricegf 03-05 08:54       ├── BreakEnigmaFile.java
.rw-rw-r--   898 ricegf 03-05 08:49       └── EncryptedPair.java
ricegf@antares:~/dev/202501/P07/baseline$ ant
Buildfile: /home/ricegf/dev/202501/P07/baseline/build.xml

build:
    [javac] Compiling 7 source files

BUILD SUCCESSFUL
Total time: 0 seconds
ricegf@antares:~/dev/202501/P07/baseline$ time java turing/BreakEnigmaFile 10
....,....,

VERIFY checksum of all decryptions is 970966332

real    0m0.109s
user    0m0.226s
sys     0m0.036s
ricegf@antares:~/dev/202501/P07/baseline$
```

WARNING: PowerShell does NOT have a `time` command. If your shell lacks a `time` command, use a search or AI engine to find something equivalent. The Hints section has suggestions.

Calibrating

Move **Results.txt** from your full_credit directory to your P07 directory.

Now increase the number of strings to break (the first parameter) until the runtime is between 45 seconds and a minute. Add this number to the **Results.txt** file and **use it for the rest of this assignment**. In the case of my Antares workstation, 76 works well.



Coding

MODIFY turing/BreakEnigmaFile.java (and ONLY this file!) to use the (optional) *second* parameter, the number of threads, to split up the work among the requested number of Java threads.

Only modify code between the delineated sections inside the `// =======` markers!

The code contains extensive comments to help you.

You'll modify one large section to instance and manage the threads, and a second very small section to protect against data contention. You may add any private fields you need at the top.

Once your code is threaded, your output should look something like the next page.

Analyzing

Once threaded, measure and record in the Full Credit section of **Results.txt** the runtimes for 1, 2, 3, ..., 16 threads (the second parameter), and answer the other Full Credit questions.

```
ricegf@antares:~/dev/202501/P07/full_credit$ time java turing.BreakEnigmaFile 76 16
+++ Starting thread 11
+++ Starting thread 10
+++ Starting thread 8
+++ Starting thread 0
+++ Starting thread 4
+++ Starting thread 14
+++ Starting thread 15
+++ Starting thread 3
+++ Starting thread 13
+++ Starting thread 12
+++ Starting thread 1
+++ Starting thread 7
+++ Starting thread 9
+++ Starting thread 6
+++ Starting thread 2
+++ Starting thread 5
### Finished thread 15
### Finished thread 4
### Finished thread 12
### Finished thread 9
### Finished thread 2
### Finished thread 14
### Finished thread 1
### Finished thread 8
### Finished thread 11
### Finished thread 0
### Finished thread 13
### Finished thread 3
### Finished thread 7
### Finished thread 6
### Finished thread 5
### Finished thread 10

-1722095701

real    0m12.488s
user    0m58.707s
sys     0m0.320s
ricegf@antares:~/dev/202501/P07/full_credit$
```

Bonus

Copy the code from cse1325-prof/P07/baseline to your own cse1325/P07/bonus directory and build it.

Now, modify ONLY file turing/BreakEnigmaFile.java to implement a *thread pool*.

That is, instead of having your threads interleave encrypted strings to break, have each thread **ask a thread pool manager method** that you write (guard against data contention!) to give it the index of the next encrypted string to break.

Once all encrypted strings have been allocated to threads, begin returning -1 to indicate that the worker threads should exit.

Measure and record in the Bonus section of **Results.txt** the runtimes for 1, 2, 3, ..., 16 threads (the second parameter), and answer the other Bonus questions.

Of particular interest is whether the thread pool approach is *faster* and / or *easier to code* that the forced allocation we used in Full Credit. We'll discuss in class - your opinion matters!

Hints

Pick a Worthy Environment

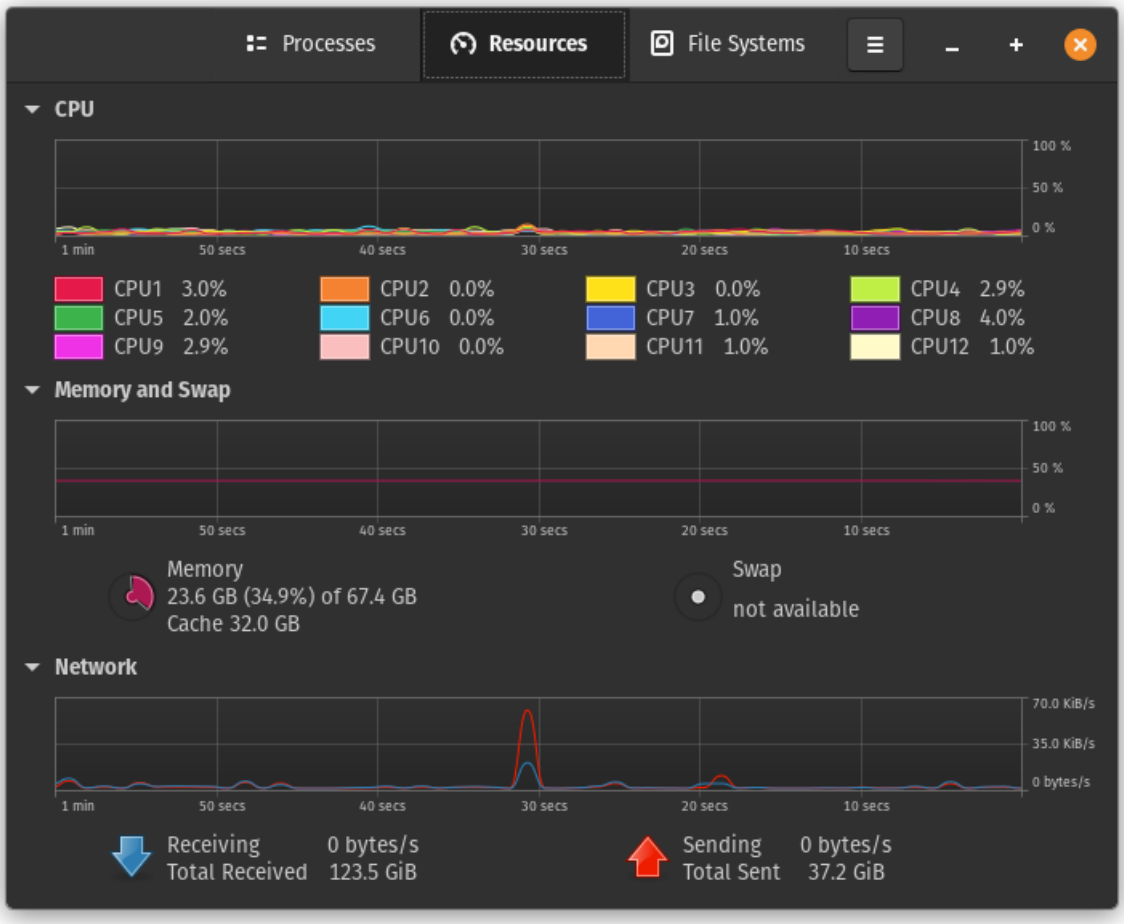
While you will receive all earned credit even on a simple, single-core machine, your results will be *much* more interesting in a multi-core environment. You may choose to develop in your host environment, which likely has many cores. You may also use a multi-core on-line environment. If you are using the CSE-VM environment and your host machine has lots of cores, you should allocate additional cores to your VM. Whatever environment offers the most cores is most desirable for this assignment.

You'll get the most consistent results if the machine on which you measure performance isn't running other heavy loads. But even a machine dedicated to testing your threaded application will show *some* variation in elapsed time.

Calibrate the Code Run Time to your Machine

Build and run the code in your **P07/full_credit** directory.

First ensure that your System Performance Monitor shows you machine is near idle. Your monitor may vary.



Then use the following command to see how long YOUR machine takes to break the codes.

```
time java turing/BreakEnigmaFile 10
```

(On a Mac, you may need to adjust the `TIMEFMT` variable - see https://unix.stackexchange.com/questions/453338/how-to-get-execution-millisecond-time-of-a-command-in-zsh. On Windows, git bash should have the `time` command pre-installed, otherwise you'll need to find a tool that measures the time a command takes to execute under that shell.)

Your output should look something like this (your format may vary):

```
ricegf@antares$ time turing/BreakEnigmaFile 50

[status messages redacted]

real   0m24.093s
user   0m25.542s
sys    0m0.273s
```

In this example, the `real` time is the "wall clock time" - that is, what the user experiences. **This is our primary focus for this assignment.** The `user` time is the sum of all of the threads, so once you get threads implemented, it may be *much* higher. The `sys` time is how much operating system overhead was consumed managing the program.

Based on the `real` time, guess how many encrypted strings you would would need to break to run for about 50 seconds. Each additional string in general takes a little longer to break - by the time you get to 120, you'll likely take hours or even days per string! So search in increments of perhaps 5 strings at a time.

*Your number of encrypted strings to break will be different than mine**, but once you find a number that makes the code run midway between 45 and 60 seconds, record it in **Results.txt** and **use that number for the rest of the assignment!**

Use git Correctly!

As always, add, commit, and push all files OFTEN!

Bonus Hints

Dividing the work arbitrarily among threads may not be the most efficient approach. What if instead of assigning an interleaved range of strings to each thread, we allow each thread to *ask for the next string* as soon as it finishes solving the current one? This keeps all threads working until all encrypted strings are assigned to be broken.

To do this, copy the baseline solution to the bonus directory (my recommendation - the suggested solution for the bonus is actually closer to the baseline code, although you may start with the full credit code if you prefer).

Then modify your thread function (within a thread-safe context) to select a board from the `encrypteds` ArrayList at the top of its main loop using a shared counter, and increment the counter within the thread-safe context. Once that string has been broken, check that shared counter to find the next string. Once the shared counter is pointing beyond the end of the ArrayList, all work has been allocated, so return -1 to indicate that the thread should exit.

Always verify that you're getting *exactly* the same checksum for the decrypted versions of the same set of encrypted strings as the baseline code provided, given the number of encrypted strings you will use for ALL aspects of this assignment.

Once everything works, complete the Bonus section of **results.txt**.

This is a simple example of a *thread pool*, in which your *worker threads* take work from a counter-based `encryptedsIndex` index *work queue* and add the checksum of each decrypted string to the `hashCodeSum` *result checksum* (though these aren't literal queues, of course). You can learn more at https://en.wikipedia.org/wiki/Thread_pool.