CSc 345 — Analysis of Discrete Structures
Spring 2013 (McCann)

http://www.cs.arizona.edu/classes/cs345/spring13/

Program #1: Somewhat Simplified Solitaire Encryption Algorithm: Two-Suit Variant

Due Date: January 31st, 2013, at the beginning of class

Overview: It will take us a little while to cover enough new material to have a 'real' programming assignment. In the meantime, I want to give you a chance to re–discover your programming skills.

In Neal Stephenson's novel "Cryptonomicon," two of the main characters are able to covertly communicate with one another with a deck of playing cards (including the jokers) and knowledge of the Solitaire encryption algorithm, which was created (in real life) by Bruce Schneier. The novel includes the description of the Solitaire algorithm in an appendix, but you can also find a revised version on the web (see below).

Assumptions for Our Modified Solitaire Algorithm: For this assignment, we'll simplify/modify the original algorithm in several ways. The biggest adjustment is in the deck being used. We'll use just two suits of a standard 52-card deck. Specifically, we'll have a total of 28 cards: The cards from the two suits (say, clubs and diamonds), plus two jokers. Further, we'll assume that the values of the 26 suit cards are 1 to 26 (i.e., Ace to King of clubs, followed by Ace to King of diamonds) The jokers ('A' and 'B') are assigned the numbers 27 and 28, respectively. Thus, the 3 of diamonds is card number 16.

The core of Solitaire is the keystream value generation algorithm. Whether we are encrypting or decrypting a message, we need one keystream value per letter in the message. Here are the steps of our modified algorithm, assuming that we start with a permutation of the 28 values described above:

1. Find the 'A' joker (27). Exchange it with the card beneath (after) it in the deck, to move the card down the deck by one position. (What if the joker is the last card in the deck? Imagine that the deck of cards is continuous; the card following the bottom card is the top card of the deck, and you'd just exchange them.)
2. Find the 'B' joker (28). Move it two cards down by performing two exchanges. Again, treat the deck as circular.
3. Swap the sequence of cards ahead of the first joker (the one closest to the top of the deck) with the sequence of cards following the second joker. This is called a *triple cut*.
4. Take the bottom card from the deck. Count down from the top card by a quantity of cards equal to the value of that bottom card. (If the bottom card is a joker, let its value be 27, regardless of which joker it is.) Take that sequence of cards and move them to the bottom of the deck. Return the bottom card to the bottom of the deck.
5. (Last step!) Look at the top card's value (which is again constrained to the range 1-27, as in the previous step). Put the card back on top of the deck. Count down the deck by that many cards. Record the value of the NEXT card in the deck, but don't remove it from the deck. If that next card happens to be a joker, don't record anything. Leave the deck the way it is, and start again from the first step, repeating until that next card is not a joker.

The value that you recorded in the last step is one value of the keystream, and will be in the range 1 – 26, inclusive (to match with the number of letters in the alphabet). To generate another keystream value, we take the deck as it is ordered after the last step and repeat the algorithm. Keep in mind that we need to generate as many keystream values as there are letters in the message being encrypted or decrypted.

As usual, an example will really help make sense of the algorithm. Let's say that this is the original ordering of our half–deck of cards:

```
AC 4C 7C 10C KC 3D 6D 9D QD JB 3C 6C 9C QC 2D 5D 8D JD JA 2C 5C 8C JC AD 4D 7D 10D KD
```

(JA and JB are the jokers), which looks like this when we convert ranks/suits to integers as described above:

```
1 4 7 10 13 16 19 22 25 28 3 6 9 12 15 18 21 24 27 2 5 8 11 14 17 20 23 26
```

Step 1: Swap the 'A' joker (27) with the value following it. So, we swap 27 and 2:

```
1 4 7 10 13 16 19 22 25 28 3 6 9 12 15 18 21 24 2 27 5 8 11 14 17 20 23 26
                                                 ^^^^
```

Step 2: Move the 'B' joker (28) two places down the list by performing two exchanges. It ends up between 6 and 9:

```
1 4 7 10 13 16 19 22 25 3 6 28 9 12 15 18 21 24 2 27 5 8 11 14 17 20 23 26
                           ^^^^^^
```

Step 3: Do the triple cut. Everything above the first joker (28 in this example) goes to the bottom of the deck, and everything below the second (27) goes to the top:

```
5 8 11 14 17 20 23 26 28 9 12 15 18 21 24 2 27 1 4 7 10 13 16 19 22 25 3 6
^^^^^^^^^^^^^^^^^^^^                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Step 4: The bottom card is 6. The first 6 cards of the deck are 5, 8, 11, 14, 17, and 20. They go just ahead of 6 at the bottom end of the deck:

```
23 26 28 9 12 15 18 21 24 2 27 1 4 7 10 13 16 19 22 25 3 5 8 11 14 17 20 6
                                                    ^^^^^^^^^^^^^^^
```

Step 5: The top card is 23. Thus, our generated keystream value is the 24^{th} card, which is 11.

Self Test: What is the next keystream value? The answer is provided at the end of this document.

OK, so what do you do with your sequence of keystream values? The answer depends on whether you are encoding a message or decoding one.

To encode a message with Solitaire, remove all non–letters and convert any lower–case letters to upper–case. (To be consistent with traditional cryptographic practice, we'll pad with the letter 'X' if necessary to have a total number of letters that is a multiple of 5.) Convert the letters to numbers (A=1, B=2, etc.). Use our keystream algorithm to generate the same number of values as are in the message. Add the corresponding pairs of numbers, modulo 26. Convert the numbers back to letters, and you've created the encrypted message.

Decryption is just the reverse of encryption. Start by converting the message to be decoded to numbers. Using the same card ordering that was used to encrypt the message originally, generate enough keystream values. (Because the same starting deck of cards was used, the same keystream sequence will be generated.) Subtract the keystream values from the message numbers, again modulo 26. Finally, convert the numbers to letters and read the message.

Let's give it a try. The message to be sent is this:

```
Dr. McCann is insane!
```

Removing the non-letters and capitalizing gives us:

```
DRMCCANNISINSANE
```

The message has 16 letters, and 16 is not a multiple of 5. To reach 20, we'll pad out the message with four X's. Next, convert the letters to numbers:

```
D  R  M  C  C  A  N  N  I  S  I  N  S  A  N  E  X  X  X  X
4 18 13  3  3  1 14 14  9 19  9 14 19  1 14  5 24 24 24 24
```

Rather than actually generating a sequence of 20 keystream values for this example, let's just pretend that we did:

```
21 6 2 19 15 18 12 23 23 5 1 7 14 6 13 1 26 16 12 20
```

Just add the two groups together pairwise. To get the modulo 26: If the sum of a pair is greater than 26, just subtract 26 from it. For example, 14 + 12 = 26, but 14 + 23 = 37 - 26 = 11. (Note that this isn't quite the result that the operator % would give you in Java.)

```
    4 18 13  3  3  1 14 14  9 19  9 14 19  1 14  5 24 24 24 24
+ 21  6  2 19 15 18 12 23 23  5  1  7 14  6 13  1 26 16 12 20
------------------------------------------------------------
  25 24 15 22 18 19 26 11  6 24 10 21  7  7  1  6 24 14 10 18
```

And convert back to letters to produce the encrypted message, which is:

```
YXOVRSZKFXJUGGAFXNJR
```

Here's how the recipient would decrypt this message. Convert the encrypted message's letters to numbers, generate the same keystream (by starting with the same deck ordering as was used for the encryption), and subtract the keystream values from the message numbers. To deal with the modulo 26 this time, just add 26 to the top number if it is equal to or smaller than the bottom number.

```
  25 24 15 22 18 19 26 11  6 24 10 21  7  7  1  6 24 14 10 18
- 21  6  2 19 15 18 12 23 23  5  1  7 14  6 13  1 26 16 12 20
------------------------------------------------------------
   4 18 13  3  3  1 14 14  9 19  9 14 19  1 14  5 24 24 24 24
```

Finally, convert the numbers to letters, and voilà: Another accurate medical diagnosis!

```
4 18 13  3  3  1 14 14  9 19  9 14 19  1 14  5 24 24 24 24
D  R  M  C  C  A  N  N  I  S  I  N  S  A  N  E  X  X  X  X
```

Assignment: Write **two** complete, well–documented, and suitably object–oriented programs named `Encrypt.java` and `Decrypt.java` that encrypt and decrypt (respectively) messages using the modified Solitaire algorithm described above. Each program will need to read in a half-deck of cards from one text file and a sequence of messages from another. Output is to be displayed to the screen ('stdout'). Note that if the message file has multiple messages to encrypt/decrypt, all but the first will be decrypted using the deck as it exists after the decryption of the preceding message. (The first uses the deck as provided, of course.)

Input: Deck files will be formatted as previously shown in this handout (see the example). The deck values will be on one line, one pair or triple of characters card, case-insignificant, with values separated by at least one space. A sample deck file, `prog1deck.dat`, is available from the class web page. Message files will have one message per line.

Your program is to accept the names of the deck and message files from the command line, in that order. (For example: `java Encrypt prog1deck.dat messages.txt`). Please note that the TAs will be running your programs on deck and message files of their own creation. You should do the same, to ensure that your algorithms are (more than!) adequately tested.

If either the deck or message files contents are invalid, display a suitable error message to the screen. Examples of possible errors include invalid suit characters in a deck file, or an encrypted message that is not a multiple of five in length.

Output: As mentioned above, the output of your programs is to be displayed to the screen. Please display one message per line. You should not attempt to add spaces when decrypted messages are displayed; just leave them without punctuation, as shown in the example above.

Hand In: On the due date, submit your well-documented program file(s) using the `turnin` facility on lectura. The submission folder is `cs345p1`. (Need help with turnin? Instructions are available from the class web page. For that matter, the documentation we expect to see is summarized in the Programming Style handout you received, and documentation examples can be found from the class web page.) Name your main program source files `Encrypt.java` and `Decrypt.java` so that we don't have to guess which files to compile, but feel free to split up your code over additional files. For instance, a keystream algorithm class is worth considering. I recommend that you submit an early version of your assignment well in advance of the due date, just to be sure that you have figured out how to use `turnin` successfully.

Want to Learn More?

- The original Solitaire algorithm (not our version!) is described here: `http://www.schneier.com/solitaire.html`.

Other Requirements and Hints:

- Start early! There are lots of little things that need to be done to complete this assignment. You may not be able to complete all of them if you wait until the night before the due date to get started.
- Make sure that you understand our modified Solitaire algorithm before you start writing the program; you can't write a program to solve a problem you don't understand.
- Don't try to write the whole assignment at once; start small. For example, you're going to have to read the initial deck from the deck file and represent it within a suitable data structure. Write that code and test it thoroughly. Then move on.
- Don't be afraid to check your program's work by hand. Take a deck file, manually generate the first few keystream letters, and check that your program generated the same ones.
- Feel free to exchange decks and encrypted messages with your classmates. Why? If you only test with your own encryption and decryption routines, any logical error with the keystream algorithm implementation is likely to appear in both routines. Without independent verification, it may appear that your logically–flawed code is correct.

Answer to the Self Test: After Step 1:

```
23 26 28 9 12 15 18 21 24 2 1 27 4 7 10 13 16 19 22 25 3 5 8 11 14 17 20 6
```

After Step 2:

```
23 26 9 12 28 15 18 21 24 2 1 27 4 7 10 13 16 19 22 25 3 5 8 11 14 17 20 6
```

After Step 3:

```
4 7 10 13 16 19 22 25 3 5 8 11 14 17 20 6 28 15 18 21 24 2 1 27 23 26 9 12
```

After Step 4:

```
14 17 20 6 28 15 18 21 24 2 1 27 23 26 9 4 7 10 13 16 19 22 25 3 5 8 11 12
```

After Step 5:

The deck is the same as it was after step 4. The 15^{th} card, the next keystream value, is 9.