- These are the files you should be submitting
  - `Deck.java`
  - `SolitaireCipher.java`

- **Do not change any of the starter code that is given to you. Add code only where instructed, namely in the "ADD CODE HERE" block.** You may add helper methods as long as you keep them `private`.

- Later next week we will also post a `Minitester` class that you can run to test if your methods are correct. This class will be equivalent to the exposed tests on codePost. Please note that these tests are only a subset of what we will be running on your submissions. We encourage you modify and expand this class.

Introduction

In 1917, Vernam patented a cipher now called *one-time pad* encryption scheme. The point of an encryption scheme is to transform a message so that only those authorized will be able to read it. One-time pad was later (in 1949) proved to be perfectly secret. The idea behind one-time pad is that given a plaintext message of length n, a uniformly random stream of digits of length n (which is the *key*) is generated and then used to encode the message. The message is concealed by replacing each character in the plaintext, with a character obtained combining the original one with one of the digits in the given key. Of course the message can be retrieved by performing the inverse operation on the characters of the encoded message (the ciphertext). Only those with access to the key can encode and decode a message. One-time pad is perfectly secret, but it has a number of drawbacks: for it to be secure, the key is required to be as long as the message, and it can only be used once! This clearly makes the cipher not a convenient one to use. Unfortunately, it was also proven that the limitations of one-time-pad are inherent to the definition of perfect secrecy. This means that to overcome those limitations the security requirements have to be relaxed.

Stream ciphers use the same idea of one-time pad encryption scheme except that a pseudorandom sequence of digits is used as the pad instead of a random one. The idea is to use what are called 'pseudorandom generators' which given a smaller key can generate streams of pseudorandom digits.

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 and knowledge of the Solitaire encryption algorithm, which was created (in real life) by Bruce Schneier. The novel includes the description of the algorithm, but you can also find a revised version on the web[1].

The Solitaire encryption algorithm is an example of a stream cipher. The key in this case is the deck of cards in its initial configuration. If two parties, Alice and Bob, share the same deck, following the Solitaire encryption algorithm they will be able to communicate by encoding and decoding messages. Of course, the deck and its configuration (i.e. the key) has to be kept secret to achieve secrecy. To encode and decode messages, Alice and Bob use the deck to generate a pseudorandom keystream which is then used as the "pad".

Encode/Decode with Solitaire

Given a message to encode, we need to first remove all non–letters and convert any lower–case letters to upper–case. We then use the keystream of values and convert each letter to the letter obtained by shifting the original one a certain number of positions *to the right* on the alphabet. This number is the one found in the keystream in the same position as the character we are encoding.

Decryption is just the reverse of encryption. Using the same keystream that was used to generate the ciphertext, convert each letter to the letter obtained by shifting the original one the given number of positions *to the left* on the alphabet.

For example, let's say that Alice wants to send the following message:

```
Is that you, Bob?
```

Then she will first remove all the non-letters and capitalize all the remaining ones obtaining the following:

```
ISTHATYOUBOB
```

She will then generate a keystream of 12 values. We'll talk about the keystream generation in the next section, so let's assume that the keystream is the following:

```
11 9 23 7 10 25 11 11 7 8 9 3
```

[1] See https://en.wikipedia.org/wiki/Solitaire_(cipher), or https://www.schneier.com/academic/solitaire/

Finally, she can generate the ciphertext by shifting each letter the appropriate number of positions to the right in the alphabet. For example, the 'I' shifted 11 positions to the right, becomes a 'T'. The 'S' shifted 9 positions to the right becomes a 'B'. And so on! The final ciphertext will be:

```
TBQOKSJZBJXE
```

Bob, upon receiving the message, will need to generate the keystream. If Alice and Bob shared the same key and used it to generate the same number of pseudorandom values, then the keystream generated in this moment by Bob will be equal to that used by Alice to encrypt the message. All there's left for Bob to do is convert all the letters by shifting them the appropriate number of position to the left.

Generating a Keystream Using a Deck of Cards

The harder part of the Solitaire encryption algorithm is generating the keystream. The idea is to use a deck of playing cards plus two jokers (a red one and a black one). Each card is associated with a value which depends on its rank and its suit. Cards in order from Ace to King have value 1 to 13 respectively. This value can increase by a multiple of 13 depending on the suit of the card. For this section let's assume we'll use the Bridge ranking for suits: clubs (lowest), followed by diamonds, hearts, and spades (highest). So, for instance, the Ace of clubs has value 1, while the 5 of diamonds has value 18, and the Queen of spades has value 51. The jokers have a value that depends on the number of cards in the deck. If the deck has a total of 54 cards (the 52 playing cards plus the two jokers), then the jokers have value 53. If the deck has total of 28 cards, then the jokers have value 27. That is, the jokers have both the same value and this value is equal to the total number of cards in the deck minus one.

The keystream values depend solely on the deck's initial configuration. We will implement the deck as a circular doubly linked list with the cards as nodes. This means that the first card (the one on the top of the deck) is linked to the last card (the one at the bottom of the deck) and the last card is linked to the first one. As an example, let's consider a deck with 28 cards: the 13 of both clubs and diamonds, plus the two jokers. Let's also consider the following initial configuration[2]:

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

The cards are represented with their rank, followed by their suit. For example, `6C` denotes the 6 of clubs, `JD` the Jack of diamonds, and `RJ` the red joker.

Here are the steps to take to generate one value of the keystream:

1. Locate the red joker and move it one card down. (That is, swap it with the card beneath it.) If the joker is the bottom card of the deck, move it just below the top card. There is no way for it to become the first card. After this step, the deck above will look as follows:

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

[2] Note that this is the same example you find on the wikipedia page https://en.wikipedia.org/wiki/Solitaire_(cipher) where instead of the cards, they list the values.

2. Locate the black joker and move it two cards down. If the joker is the bottom card of the deck, move it just below the second card. If the joker is one up from the bottom card, move it just below the top card. There is no way for it to become the first card. After this step, the deck above will look as follows:

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

3. Perform a "triple cut": that is, swap the cards above the first joker with the cards below the second joker. Note that here we use "first" and "second" joker to refer to whatever joker is nearest to, and furthest from, the top of the deck. Their colors do not matter. Note that the jokers and the cards between them do not move! If there are no cards in one of the three sections (either the jokers are adjacent, or one is on top or the bottom), just treat that section as empty and move it anyway. The deck will now look as follows:

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

4. Perform a "count cut": look at the value of the bottom card. Remove that number of cards from the top of the deck and insert them just above the last card in the deck. The deck will now look as follows:

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

5. Finally, look at the value of the card on the top of the deck. Count down that many cards. (Count the top card as number one.) If you hit a joker, ignore it and repeat the keystream algorithm. Otherwise, use the value of the card you counted to as the next keystream value. Note that this step does not modify the state of the deck. In our example, the top card is a 10 of diamonds which has value 23. By counting down to the 24th card we find the Jack of clubs which has value 11. Hence, 11 would be the first keystream value generated by our deck.

Instructions and Starter Code

As mentioned in the section before we will use a circular doubly linked list to represent a deck of cards. The starter code contains two files with five classes which are as follows:

- `Deck` - This class defines a deck of cards. Most of your work goes into this file. This class contains three nested classes: `Card`, `PlayingCard`, and `Joker`.
- `SolitaireCipher` – This class represents a stream cipher that uses the Solitaire algorithm to generate the keystream and then encode/decode messages.

Please note that we defined all the members of the classes `public` for testing purposes. In reality, for better coding style, most of those methods and all of the fields should have been kept `private`.

Methods you need to implement

you need to implement all of the methods listed below. See the starter code for the full method signatures. Your implementations must be efficient. For each method below, we indicate the worst case run time using $O()$ notation.

- `Deck.Deck(int numOfCardsPerSuit, int numOfSuits)` : creates a deck with cards from Ace to `numOfCardsPerSuit` for the first `numOfSuits` in the class field `suitsInOrder`. The cards should be ordered first by suit, and then by rank. In addition to these cards, a red joker and a black joker are added to the bottom of the deck in this order. For example, with input 4 and 3, and `suitsInOrder` as specified in the file, the deck contains the following cards **in this specific order**:

  ```
  AC 2C 3C 4C AD 2D 3D 4D AH 2H 3H 4H RJ BJ
  ```

  The constructor should raise an `IllegalArgumentException` if the first input is not a number between 1 and 13 (both included) or the second input is not a number between 1 and the size of the class field `suitsInOrder`. Remember that a deck is a **circular doubly linked list** so make sure to set up all the pointers correctly, as well as the instance fields.

- `Deck.Deck(Deck d)` : creates a deck by making a deep copy of the input deck. Hint: use the method `getCopy` from the class `Card`. *Disclaimer: this is not the correct way of making a deep copy of objects that contain circular references, but it is a simple one and good enough for our purposes.*

- `Deck.addCard(Card c)` : adds the input card to the bottom of the deck. This method runs in $O(1)$.

- `Deck.shuffle()` : shuffles the deck. There are different ways of doing this, but for this assignment you will need to implement an algorithm that uses the Fisher–Yates shuffle algorithm. The algorithm runs in $O(n)$ using $O(n)$ space, where n is the number of cards in the deck. To perform a shuffle of the deck follow the steps:

  - Copy all the cards inside an array
  - Shuffle the array using the following algorithm:

    ```
    for i from n-1 to 1 do
        j <-- random integer such that 0 <= j <= i
        swap a[j] and a[i]
    ```

    To generate a random integer use the `Random` object stored in the class field called `gen`.

  - Use the array to rebuild the shuffled deck.

- `Deck.locateJoker(String color)` : returns a reference to the joker in the deck with the specified color. This method runs in $O(n)$.

- `Deck.moveCard(Card c, int p)` : moves the card `c` by `p` positions down the deck. You can assume that the input card belongs to the deck (which implies that the deck is not empty). This method runs in $O(p)$.

- `Deck.tripleCut(Card firstCard, Card secondCard)` : performs a triple cut on the deck using the two input cards. You can assume that the input cards belong to the deck and the first one is nearest to the top of the deck. This method runs in $O(1)$.

- `Deck.countCut()` : performs a count cut on the deck. The number used for the cut is the value of the bottom card modulo the total number of cards in the deck. Note that this means that if the value of the bottom card is equal to a multiple of the number of cards in the deck, then the method should not do anything. This method runs in $O(n)$.
- `Deck.lookUpCard()` : returns a reference to the card that can be found by looking at the value of the card on the top of the deck, and counting down that many cards. If the card found is a Joker, then the method returns `null`, otherwise it returns the card found. This method runs in $O(n)$.
- `Deck.generateNextKeystreamValue()` : uses the Solitaire algorithm to generate one value for the keystream using this deck. This method runs in $O(n)$.
- `SolitaireCipher.getKeystream(int size)` : generates a keystream of the given size.
- `SolitaireCipher.encode(String msg)` : encodes the input message by generating a keystream of the correct size and using it to encode the message as described earlier in the pdf.
- `SolitaireCipher.decode(String msg)` : decodes the input message by generating a keystream of the correct size and using it to decode the message as described earlier in the pdf.

Small example

Generate a deck of 12 cards as follows:

`AC 2C 3C 4C 5C AD 2D 3D 4D 5D RJ BJ`

If you seed the random generator using 10 as the seed, then after shuffling the deck once you will get the following configuration:

`3C 3D AD 5C BJ 2C 2D 4D AC RJ 4C 5D`

If you were to use this deck to create a Solitaire cipher and you would try to encode the message `"Is that you, Bob?"` you would get the ciphertext `MWIKDVZCKSFP` obtained using the following keystream:

`4 4 15 3 3 2 1 14 16 17 17 14`

Finally, note that the keystream used in the section describing how to encode/decode is the keystream you would obtain using the deck from the example used on how to generate keystream values.