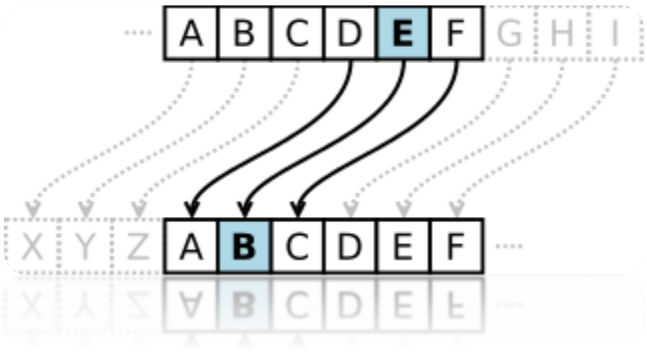
IST 451 Homework 1

Option 2 – Programming



Timothy Flynn

January 25, 2015

Part A: Code and Output

Code

```java
package caesarcipher;
/**
 *@author Timothy Flynn
 *Code modified from source provided at: http://rosettacode.org/wiki/Caesar_cipher#Java
 */
public class CaesarCipher {
    public static void main(String[] args) {
    //Declare and initialize String givenString with the appropriate value (from assignment)
    String givenString = "ZSAD LG LZW DAGF DGQSD SFV LJMW ZSAD SDES ESLWJ OALZ QGMJ OZALW SFV TDMW";
    //Print out original ciphertext
    System.out.println("Ciphertext: \t" + givenString +"\n");
    //Declare String result for use later
    String result;
    //Loop through all 26 combinations of the Caesar Cipher
    for(int i = 1; i <= 26; i++)
    {
        //Store the value of the encode with the given offset in result
        result = encode(givenString, i);
        //Display result with some formatting
        System.out.println("Offset " + i + ":\t" + result);
        //Loop again with new offset
     }
    }
    //Method for encoding Caesar Cipher
    //Takes two arguments, String enc to be encoded and int offset for number of characters to offset
    public static String encode(String enc, int offset) {
        //Take the offset mod 26 then add 26
        offset = offset % 26 + 26;
        //Declare a StringBuilder to access individual parts of the String
        StringBuilder encoded = new StringBuilder();
        //Iterate through all characters in String enc
        for (char i : enc.toCharArray()) {
            //Nested ifs to shift individual elements of the character array
            if (Character.isLetter(i)) {
                //Check for upper case/lower case
                if (Character.isUpperCase(i)) {
                    //Append character + offset (taking into account wraparound)
                    encoded.append((char) ('A' + (i - 'A' + offset) % 26 ));
                }
                //Append character + offset (taking into account wraparound)
                else {
                    encoded.append((char) ('a' + (i - 'a' + offset) % 26 ));
                }
            } else {
                //Append characcter to StringBuilder object
                encoded.append(i);
            }
        }
        //Return encoded string
        return encoded.toString();
    }
}
```

Output

Upon execution, the program will solve the original problem, which was to display all possible shift-ciphers that may have been used to create the ciphertext ZSAD LG LZW DAGF DGQSD SFV LJMW ZSAD SDES ESLWJ OALZ QGMJ OZALW SFV TDMW.

The following output is obtained when executing the program:

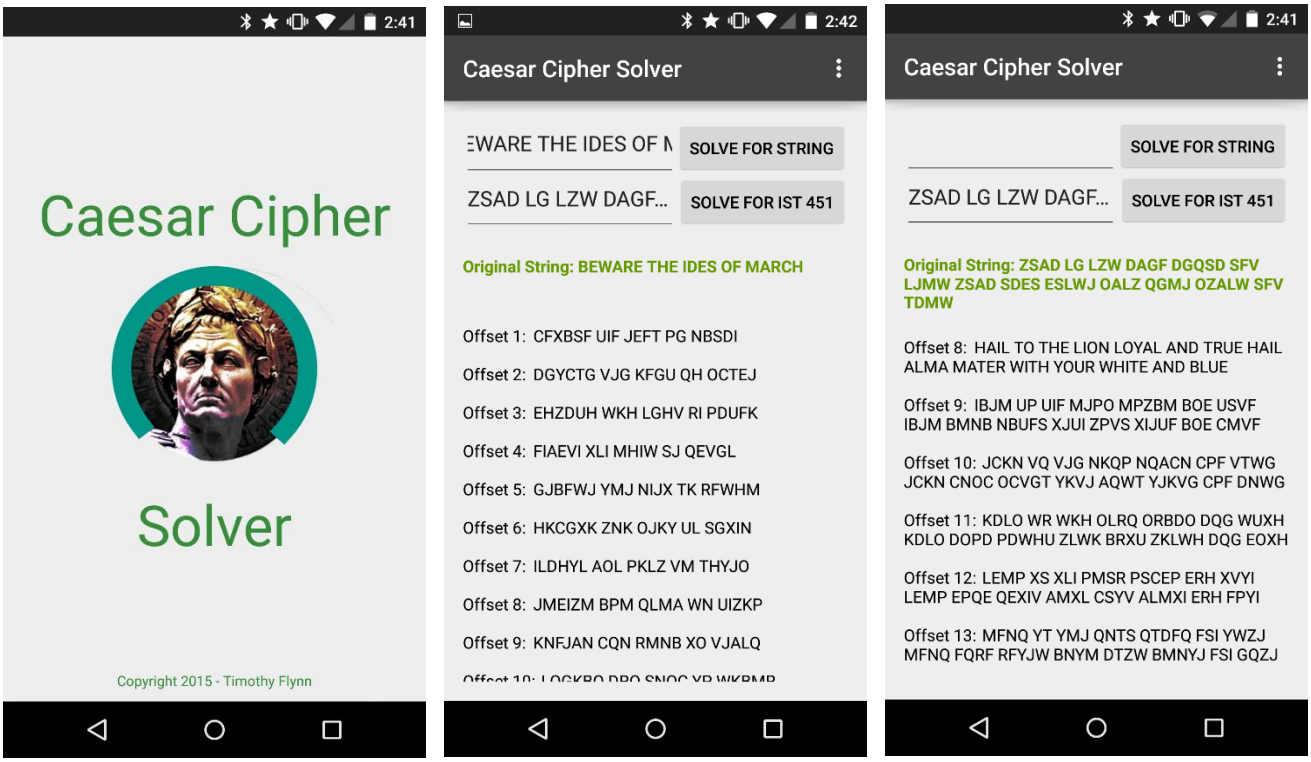
```
Ciphertext:        ZSAD LG LZW DAGF DGQSD SFV LJMW ZSAD SDES ESLWJ OALZ QGMJ OZALW SFV TDMW

Offset 1:          ATBE MH MAX EBHG EHRTE TGW MKNX ATBE TEFT FTMXK PBMA RHNK PABMX TGW UENX
Offset 2:          BUCF NI NBY FCIH FISUF UHX NLOY BUCF UFGU GUNYL QCNB SIOL QBCNY UHX VFOY
Offset 3:          CVDG OJ OCZ GDJI GJTVG VIY OMPZ CVDG VGHV HVOZM RDOC TJPM RCDOZ VIY WGPZ
Offset 4:          DWEH PK PDA HEKJ HKUWH WJZ PNQA DWEH WHIW IWPAN SEPD UKQN SDEPA WJZ XHQA
Offset 5:          EXFI QL QEB IFLK ILVXI XKA QORB EXFI XIJX JXQBO TFQE VLRO TEFQB XKA YIRB
Offset 6:          FYGJ RM RFC JGML JMWYJ YLB RPSC FYGJ YJKY KYRCP UGRF WMSP UFGRC YLB ZJSC
Offset 7:          GZHK SN SGD KHNM KNXZK ZMC SQTD GZHK ZKLZ LZSDQ VHSG XNTQ VGHSD ZMC AKTD
Offset 8:          HAIL TO THE LION LOYAL AND TRUE HAIL ALMA MATER WITH YOUR WHITE AND BLUE
Offset 9:          IBJM UP UIF MJPO MPZBM BOE USVF IBJM BMNB NBUFS XJUI ZPVS XIJUF BOE CMVF
Offset 10:         JCKN VQ VJG NKQP NQACN CPF VTWG JCKN CNOC OCVGT YKVJ AQWT YJKVG CPF DNWG
Offset 11:         KDLO WR WKH OLRQ ORBDO DQG WUXH KDLO DOPD PDWHU ZLWK BRXU ZKLWH DQG EOXH
Offset 12:         LEMP XS XLI PMSR PSCEP ERH XVYI LEMP EPQE QEXIV AMXL CSYV ALMXI ERH FPYI
Offset 13:         MFNQ YT YMJ QNTS QTDFQ FSI YWZJ MFNQ FQRF RFYJW BNYM DTZW BMNYJ FSI GQZJ
Offset 14:         NGOR ZU ZNK ROUT RUEGR GTJ ZXAK NGOR GRSG SGZKX COZN EUAX CNOZK GTJ HRAK
Offset 15:         OHPS AV AOL SPVU SVFHS HUK AYBL OHPS HSTH THALY DPAO FVBY DOPAL HUK ISBL
Offset 16:         PIQT BW BPM TQWV TWGIT IVL BZCM PIQT ITUI UIBMZ EQBP GWCZ EPQBM IVL JTCM
Offset 17:         QJRU CX CQN URXW UXHJU JWM CADN QJRU JUVJ VJCNA FRCQ HXDA FQRCN JWM KUDN
Offset 18:         RKSV DY DRO VSYX VYIKV KXN DBEO RKSV KVWK WKDOB GSDR IYEB GRSDO KXN LVEO
Offset 19:         SLTW EZ ESP WTZY WZJLW LYO ECFP SLTW LWXL XLEPC HTES JZFC HSTEP LYO MWFP
Offset 20:         TMUX FA FTQ XUAZ XAKMX MZP FDGQ TMUX MXYM YMFQD IUFT KAGD ITUFQ MZP NXGQ
Offset 21:         UNVY GB GUR YVBA YBLNY NAQ GEHR UNVY NYZN ZNGRE JVGU LBHE JUVGR NAQ OYHR
Offset 22:         VOWZ HC HVS ZWCB ZCMOZ OBR HFIS VOWZ OZAO AOHSF KWHV MCIF KVWHS OBR PZIS
Offset 23:         WPXA ID IWT AXDC ADNPA PCS IGJT WPXA PABP BPITG LXIW NDJG LWXIT PCS QAJT
Offset 24:         XQYB JE JXU BYED BEOQB QDT JHKU XQYB QBCQ CQJUH MYJX OEKH MXYJU QDT RBKU
Offset 25:         YRZC KF KYV CZFE CFPRC REU KILV YRZC RCDR DRKVI NZKY PFLI NYZKV REU SCLV
Offset 26:         ZSAD LG LZW DAGF DGQSD SFV LJMW ZSAD SDES ESLWJ OALZ QGMJ OZALW SFV TDMW
```

With some careful investigation, it becomes obvious that the plaintext that was originally encoded was `Offset 8: HAIL TO THE LION LOYAL AND TRUE HAIL ALMA MATER WITH YOUR WHITE AND BLUE`. This program makes it much easier to see all possible combinations and therefore decode the encrypted message.

Android Application Port

After successfully implementing a solution in Java, I decided it would be interesting to port the application to Android to make it a little more user friendly. Due to the fact that Android runs Java, the algorithm stayed mostly unchanged, and any additional code was used to implement different user controls (buttons, text fields, etc.). I also decided to add functionality that would allow users to input their own ciphertext to be decoded.



Implementing the project in Android was quite interesting and definitely was a useful learning experience (even if somewhat unrelated to IST 451). I am currently working on Android projects for a few other classes and this was a nice exercise in some of the basics of application design and development.

Part B: Logic Explanation

The code in part A consists of two relatively simple parts, a function that performs the shift-cipher (lines 24-51) and a loop that executes this function for every possible combination (lines 15-22). These two pieces of code are put together to display every possible shift-cipher combination.

Encoding Function

The encoding function asks for two pieces of information before being executed (these pieces of information are called parameters). These parameters are used inside of the function to produce the correct output. The first parameter is our ciphertext that we wish to encode (in the program it takes the name of `String` **`enc`**). The second parameter is the shift we wish to apply to the ciphertext (in the code this is stored as `int` **`offset`**).

Once this information is provided to the function, the calculation must be performed. The function first takes the offset and performs the modulus function on it (included below).

$$E_n(x) = (x + n) \mod 26.$$

What this function does is ensure that the offset value falls from 0-25. The program then takes these 26 possible numbers and maps them to the 26 available letters, so that the transformation can be applied mathematically and displayed visually.

After the math is done for the offset, the program must define a variable that will represent our encoded text. In this program, this takes the form of `StringBuilder encoded`, which will allow the program to have a piece of text in which each individual letter can be modified.

After the variable is set for the encoded value, we must go through every single letter and perform the shift. This is accomplished using a for-loop, a type of loop that will execute the same piece of code for every letter in our provided string.

For every letter, the program will do a few things. First, it will check to see if the current character is a letter, then it will check whether it is upper or lower case, and finally it will perform the shift then add the current character to the variable `encoded` which we defined earlier.

The first few steps are relatively simple with functionality provided by Java, but applying the shift takes more code. To apply the shift, the program takes the current character (stored in the variable `i`) and performs the shift function on it. The shift function takes the first letter in the alphabet as a reference (in our case A) and then adds it to the difference of the provided character and A plus the offset. This modulus function is once again run to ensure that the value falls between 0-25 and therefore is represented by a letter of the alphabet. What this accomplishes is applying the appropriate shift (as we defined earlier in **`offset`**) and making sure that letters wrap around from Z →A.

Once the appropriate shift is applied, the character is stored in a variable, and then the next character is shifted until the entire piece of provided text has been encoded.

Loop

In order to display all possible shift-cipher combinations, the aforementioned encoding function must be run 26 separate times. In order to save time and space, a loop is used. This particular loop is a for-loop that will perform a shift-cipher operation on a piece of given ciphertext 26 times.

Every time the loop is run, the encoded text is stored in a variable called `result` and then outputted to the console. With each repetition, the value of the offset is changed by one. This ensures that all possible combinations are displayed.

Part C: Enhancements

This program is relatively basic in regards to functionality and could easily be enhanced in order to make it more useful to users. Some possible ways of doing this include having the program automatically select the most likely candidate for the decryption, and doing a similar brute-force attack on a mono-alphabetic cipher with a randomly ordered alphabet.

Most Likely Candidate Detection

In order to detect the most likely candidate for the decryption, several different methods could be taken. The method that I would most likely attempt would be language detection. With proper language detection configured, the program could determine which versions of the ciphertext are comprised of actual words and filter them out from the other candidates. Methods for determining the language of a given piece of text can use statistical analysis, letter frequency, and n-grams to determine the language of a given piece of text.

There are many language detection algorithms that have been developed and would be able to enhance the program. One of them is a Java library called *language-detection* (Shuyo, 2014) that uses Wikipedia to create language profiles which are then used to detect languages. It is touted to have a 99% accuracy for 53 languages which would hopefully catch most of the likely candidates.

Randomly Ordered Mono-alphabetic Cipher Functionality

In order to perform a brute force attack on a randomly ordered mono-alphabetic cipher, many more combinations would need to be attempted. This would increase both the complexity and computing power required to perform the task. In order to perform an exhaustive key search, all random combinations of the alphabet would need to be tested. The number of possible keys can be calculated using the following formula:

$$26! \approx 2^{88.4}$$

This number is quite large and it would take quite a while to try all the possible combinations. One weakness of the mono-alphabetic cipher is that the letter distribution remains intact and can be used to determine likely candidates for each letter. A program that works to decode this type of cipher in a brute-force style could use letter frequency distribution to determine which letters occur most often and compare that against the commonly occurring letters in other popular languages. When a near match is found, suggested plaintext-ciphertext pairs could be suggested and used to decrypt the message. If this did not work, the program could attempt all the possible combinations, but even at 1 trillion attempts per second, it would take a computer nearly 12 million years to complete this search (Sauerberg, 2013). With this information in mind, the most likely way of breaking a randomly ordered mono-alphabetic cipher would be to use letter frequency distributions and attempt to match those to know frequencies of common languages.

References

Sauerberg, J. (2013). Cryptology: An Historical Introduction. In J. Sauerberg, *Cryptology: An Historical Introduction* (p. 257).

Shuyo, N. (2014, March 3). *language-detection*. Retrieved from code.google.com: https://code.google.com/p/language-detection/