CSCI361 CRYPTOGRAPHY & SECURE APPLICATIONS
ASSIGNMENT 1

NAME: SALLY
UOW ID: 4603229

I HAVE READ THE POLICY FOR PLAGARISM AT UNIVERSITY OF WOLLONGONG.
I DECLARE THAT THIS ASSIGNMENT SOLUTION IS ENTIRELY MY OWN WORK.

PART 1

How to compile the program:

---

1. Open your command prompt
2. Navigate to folder where files are stored
3. javac affineCipher.java

4a. Encryption: java affineCipher -key <a> <b> -encrypt -in <input.txt> -out <output.txt>
Java affineCipher -key 3 8 -encrypt -in input.txt -out output.txt

4b. Decryption: java affineCipher -key <a> <b> -decrypt -in <input.txt> -out <output.txt>
Java affineCipher -key 3 8 -decrypt -in output.txt -out output2.txt

Take Note!
a must be prime number

In Affine Cipher, we can only have a unique pair of keys (a, b) if a is relatively prime to the mod value (in this case is 26) so to prevent different plaintext from getting the same ciphertext through encryption.
Using Euler's totient function, the numbers of a that is relatively prime to 26 are {1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, and 25}.

1(i) (3,9) is valid.

1(ii) (6,4) is invalid as 6 is an even number and share a common divisor of 2 with 26

1(iii) (11,0) is valid.

1(iv) (0,13) is invalid as 0 is not relatively prime to any number

1(v) (13,1) is invalid as 13 shares a common divisor of 13 with 26

PART 1

2. To decrypt the Affine Cipher, for each value of C of the message, correspond a value M, result of inverse function M = a' x (C – b) mod 26
The value a' is an integer such as **a x a' = 1 mod 26**
To find a' we can use Extended Euclidean Algorithm to calculate its modular inverse.

For key (3,9)

N1	N2	R	Q	A1	A2	B1	B2
26	3	2	8	1	0	0	1
3	2	1	1	0	1	1	-8
2	1	0	2	1	-1	-8	9

a' = 9
Let's say we encrypt the alphabet 'e' which is index 4 using the key (3,9).
We get the index of encrypted letter of 3(4) + 9 = 21 mod 26 = 21, which is the index of letter 'v'
If we want to decrypt 'v', we compute 9 x (21-9) mod 26 = 108 mod 26 = 4 which is the index of 'e'

For key (11,0)

N1	N2	R	Q	A1	A2	B1	B2
26	11	4	2	1	0	0	1
11	4	3	2	0	1	1	-2
4	3	1	1	1	-2	-2	5
3	1	0	3	-2	3	5	-7

a' = -7
Let's say we encrypt the alphabet 'e' which is index 4 using the key (11,0).
We get the index of encrypted letter of 11(4) + 0 = 44 mod 26 = 18, which is the index of letter 's'.
If we want to decrypt 's', we compute -7 x (18-0) mod 26 = -126 mod 26 = 4 which is the index of 'e'

PART 1

3. Encryption/Decryption Output Screenshot

```
sallyyeo@sallys-MacBook-Pro desktop % javac affineCipher.java
sallyyeo@sallys-MacBook-Pro desktop % java affineCipher -key 3 9 -encrypt -in input.txt -out output.txt

Plaintext Message is : ABCDEFGHIJKLMNOPQRSTUVWXYZ, abcdefghijklmnopqrstuvwxyz. 1234567890.

Encrypted Message is : JMPSVYBEHKNQTWZCFILORUXADG, jmpsvybehknqtwzcfiloruxadg. 1234567890.
sallyyeo@sallys-MacBook-Pro desktop % java affineCipher -key 3 9 -decrypt -in output.txt -out output2.tx
t

Encrypted Message is : JMPSVYBEHKNQTWZCFILORUXADG, jmpsvybehknqtwzcfiloruxadg. 1234567890.

Decrypted Message is: ABCDEFGHIJKLMNOPQRSTUVWXYZ, abcdefghijklmnopqrstuvwxyz. 1234567890.
```

Detect invalid key & display error message

```
sallyyeo@sallys-MacBook-Pro desktop % java affineCipher -key 2 9 -encrypt -in input.txt -out output.txt

Key a is invalid
```

4. Encryption/Decryption using key (3,9)

```
sallyyeo@sallys-MacBook-Pro desktop % javac affineCipher.java
sallyyeo@sallys-MacBook-Pro desktop % java affineCipher -key 3 9 -encrypt -in input.txt -out output.txt

Plaintext Message is : ABCDEFGHIJKLMNOPQRSTUVWXYZ, abcdefghijklmnopqrstuvwxyz. 1234567890.

Encrypted Message is : JMPSVYBEHKNQTWZCFILORUXADG, jmpsvybehknqtwzcfiloruxadg. 1234567890.
sallyyeo@sallys-MacBook-Pro desktop % java affineCipher -key 3 9 -decrypt -in output.txt -out output2.tx
t

Encrypted Message is : JMPSVYBEHKNQTWZCFILORUXADG, jmpsvybehknqtwzcfiloruxadg. 1234567890.

Decrypted Message is: ABCDEFGHIJKLMNOPQRSTUVWXYZ, abcdefghijklmnopqrstuvwxyz. 1234567890.
```

4. Encryption/Decryption using key (11,0)

```
sallyyeo@sallys-MacBook-Pro desktop % java affineCipher -key 11 0 -encrypt -in input.txt -out output.txt

Plaintext Message is : ABCDEFGHIJKLMNOPQRSTUVWXYZ, abcdefghijklmnopqrstuvwxyz. 1234567890.

Encrypted Message is : ALWHSDOZKVGRCNYJUFQBMXITEP, alwhsdozkvgrcnyjufqbmxitep. 1234567890.
sallyyeo@sallys-MacBook-Pro desktop % java affineCipher -key 11 0 -decrypt -in output.txt -out output2.txt

Encrypted Message is : ALWHSDOZKVGRCNYJUFQBMXITEP, alwhsdozkvgrcnyjufqbmxitep. 1234567890.

Decrypted Message is: ABCDEFGHIJKLMNOPQRSTUVWXYZ, abcdefghijklmnopqrstuvwxyz. 1234567890.
```

PART 2

How to compile the program:

---

1. Open your command prompt
2. Navigate to folder where java files and Assignment1-Part2.txt are stored
3. javac affineCipher.java
4. java affineCipher -decrypt -in Assignment1-Part2.txt -out output.txt

Answer: The key pair is (5, 4)
The encryption and decryption algorithm is using part 1 code (affineCipher.java)

Output Screenshot:

```
[sallyyeo@sallys-MBP desktop % javac affineCipher.java
[sallyyeo@sallys-MBP desktop % java affineCipher -decrypt -in Assignment1-Part2.txt -out output.txt
Encrypted Message is : Smbhymyrv vny srtyp wd owsrostyroy ert vny mavaeh srtyp wd owsrostyroy vw owmbavy
vny dwhhwksri:
    - vny srtyp wd owsrostyroy wd vypv E.
    - vny mavaeh srtyp wd owsrostyroy wd vypvq E ert J.

Lymymjyl vw qajmsv ehh blwilemmsri owtyq vnev uwa klwvy dwl twsri vnsq veqc.

Uwa kshh rwv jy isfyr melcq sd uwa xaqv isfy vnwqy srtsoyq ksvnwav blwbyl owtyq.

Vypv E: oujyl qyoalsvu sq ejwav nwk ky tyfyhwb qyoaly owmbavylq ert owmbavyl ryvkwlcq, vw yrqaly vnev vny
 teve qvwlyt ert vlerqmsvvyt vnlwain vnym sq blwvyovyt dlwm areavnwlszyt eooyqq wl vw owmjev tsisveh qyoa
lsvu vnlyevq ert nezeltq. eq ky owrtaov mwly wd wal qwoseh, owrqamyl ert jaqsryqq eovsfsvsyq wrhsry, vnyl
y sq e owllyqbwrtsri srolyeqy sr vny tymert dwl sov blwdyqqswrehq vw mereiy wal tsisveh yrfslwrmyrv ert y
owrwmu.

Vypv J: oujyl qyoalsvu neq jyyr styrvsdsyt eq wry wd vny qvlevyiso blswlsvsyq sr eaqvlehse vw myyv vny ty
mertq wd hek yrdwloymyrv, revswreh ert qvevy iwfylrmyrvq, tydyrqy, qyoalsvu ert dsreroy srtaqvlsyq. xwjq
wd vny davaly kshh jy sr ehh wd vnyqy elyeq yrqalsri vnyly sq revswreh oebejshsvu vw mesrvesr ert jasht w
al yqqyrvseh qylfsoyq ert qvwb vnym dlwm jysri tsqlabvyt, tyqvlwuyt, wl vnlyevyryt, ert vnev wal bylqwreh
 srdwlmevswr sq rwv owmmarsoevyt, qnelyt, fsqaehszyt wl erehuqyt ksvnwav wal bylmsqqswr.


Decrypted Message is : Implement the index of coincidence and the mutual index of coincidence to compute
the following:
    - the index of coincidence of text A.
    - the mutual index of coincidence of texts A and B.

Remember to submit all programming codes that you wrote for doing this task.

You will not be given marks if you just give those indices without proper codes.

Text A: cyber security is about how we develop secure computers and computer networks, to ensure that the
 data stored and transmitted through them is protected from unauthorized access or to combat digital secu
rity threats and hazards. as we conduct more of our social, consumer and business activities online, ther
e is a corresponding increase in the demand for ict professionals to manage our digital environment and e
conomy.

Text B: cyber security has been identified as one of the strategic priorities in australia to meet the de
mands of law enforcement, national and state governments, defense, security and finance industries. jobs
of the future will be in all of these areas ensuring there is national capability to maintain and build o
ur essential services and stop them from being disrupted, destroyed, or threatened, and that our personal
 information is not communicated, shared, visualized or analysed without our permission.


The keys are (5, 4)
The index of coincidence of text A               : 0.067
The mutual index of coincidence of text A & B    : 0.069
```

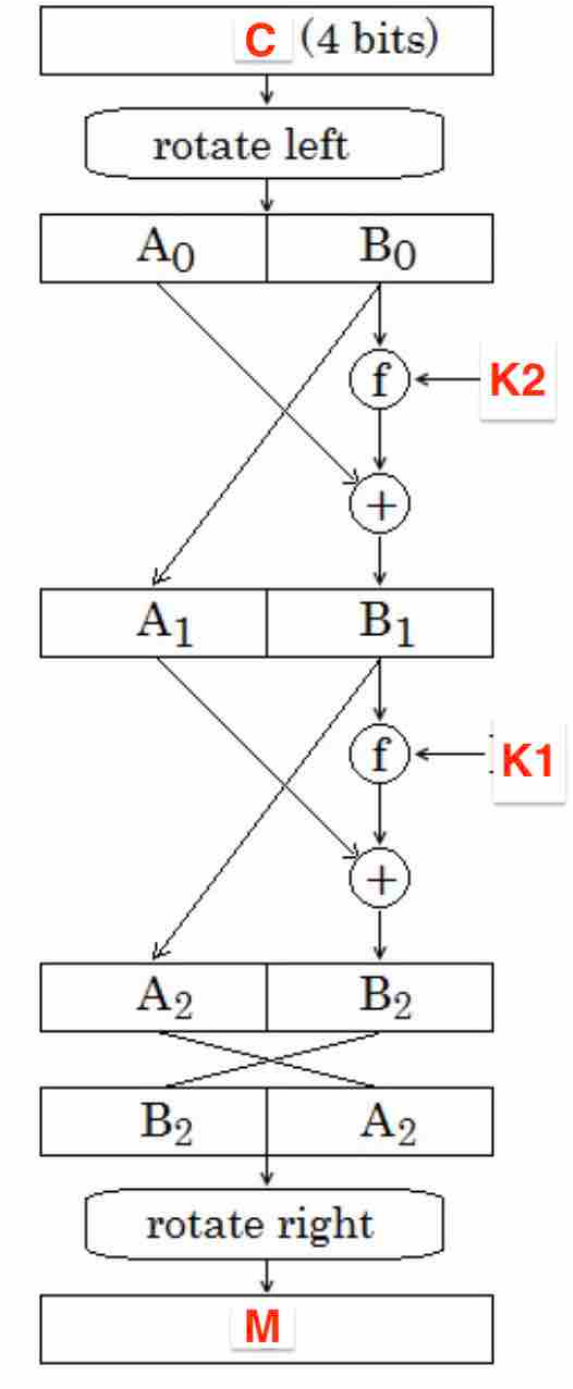
PART 3

How to compile the program:

---

1. Open your command prompt
2. Navigate to folder where java file is stored
3. javac LinearDES.java
4. java LinearDES

1) Corresponding decryption diagram



PART 3

2) Encryption & Decryption Output Screenshot

```
sallyyeo@sallys-MBP desktop % javac LinearDES.java
sallyyeo@sallys-MBP desktop % java LinearDES
LDES Encryption:
Key\Message:      0000  0001  0010  0011  0100  0101  0110  0111  1000  1001  1010  1011  1100  1101  1110  1111
   00             1001  1000  0011  0010  1100  1101  0110  0111  0001  0000  1011  1010  0100  0101  1110  1111
   01             1111  1110  0101  0100  1010  1011  0000  0001  0111  0110  1101  1100  0010  0011  1000  1001
   10             0110  0111  1100  1101  0011  0010  1001  1000  1110  1111  0100  0101  1011  1010  0001  0000
   11             0000  0001  1010  1011  0101  0100  1111  1110  1000  1001  0010  0011  1101  1100  0111  0110

LDES Decryption:
Key\Message:      0000  0001  0010  0011  0100  0101  0110  0111  1000  1001  1010  1011  1100  1101  1110  1111
   00             1001  1000  0011  0010  1100  1101  0110  0111  0001  0000  1011  1010  0100  0101  1110  1111
   01             0110  0111  1100  1101  0011  0010  1001  1000  1110  1111  0100  0101  1011  1010  0001  0000
   10             1111  1110  0101  0100  1010  1011  0000  0001  0111  0110  1101  1100  0010  0011  1000  1001
   11             0000  0001  1010  1011  0101  0100  1111  1110  1000  1001  0010  0011  1101  1100  0111  0110
```

3) Verify for every key, E(1100) = E(1000) + E(0100) + E(0000)

Output Screenshot

```
Verification:
Equation:                  1100 =  1000 ^  0100 ^  0000
Verified with key: 00->    0100 =  0001 ^  1100 ^  1001  Verified:  0100
Verified with key: 01->    0010 =  0111 ^  1010 ^  1111  Verified:  0010
Verified with key: 10->    1011 =  1110 ^  0011 ^  0110  Verified:  1011
Verified with key: 11->    1101 =  1000 ^  0101 ^  0000  Verified:  1101
```

PART 3

3) Output Screenshot for similar equations for E(1010), E(1001), E(0110), E(0101), E(0011), E(0111), E(1011), E(1101), E(1110), E(1111) in terms of E(0000), E(1000), E(0100), E(0010), E(0001).

```
Verification:
Equation:               0011 =  0010 ^  0001 ^  0000
Verified with key: 00-> 0010 =  0011 ^  1000 ^  1001  Verified:  0010
Verified with key: 01-> 0100 =  0101 ^  1110 ^  1111  Verified:  0100
Verified with key: 10-> 1101 =  1100 ^  0111 ^  0110  Verified:  1101
Verified with key: 11-> 1011 =  1010 ^  0001 ^  0000  Verified:  1011

Verification:
Equation:               1010 =  1000 ^  0010 ^  0000
Verified with key: 00-> 1011 =  0001 ^  0011 ^  1001  Verified:  1011
Verified with key: 01-> 1101 =  0111 ^  0101 ^  1111  Verified:  1101
Verified with key: 10-> 0100 =  1110 ^  1100 ^  0110  Verified:  0100
Verified with key: 11-> 0010 =  1000 ^  1010 ^  0000  Verified:  0010

Verification:
Equation:               1001 =  1000 ^  0001 ^  0000
Verified with key: 00-> 0000 =  0001 ^  1000 ^  1001  Verified:  0000
Verified with key: 01-> 0110 =  0111 ^  1110 ^  1111  Verified:  0110
Verified with key: 10-> 1111 =  1110 ^  0111 ^  0110  Verified:  1111
Verified with key: 11-> 1001 =  1000 ^  0001 ^  0000  Verified:  1001

Verification:
Equation:               0110 =  0100 ^  0010 ^  0000
Verified with key: 00-> 0110 =  1100 ^  0011 ^  1001  Verified:  0110
Verified with key: 01-> 0000 =  1010 ^  0101 ^  1111  Verified:  0000
Verified with key: 10-> 1001 =  0011 ^  1100 ^  0110  Verified:  1001
Verified with key: 11-> 1111 =  0101 ^  1010 ^  0000  Verified:  1111

Verification:
Equation:               0101 =  0100 ^  0001 ^  0000
Verified with key: 00-> 1101 =  1100 ^  1000 ^  1001  Verified:  1101
Verified with key: 01-> 1011 =  1010 ^  1110 ^  1111  Verified:  1011
Verified with key: 10-> 0010 =  0011 ^  0111 ^  0110  Verified:  0010
Verified with key: 11-> 0100 =  0101 ^  0001 ^  0000  Verified:  0100

Verification:
Equation:               0011 =  0010 ^  0001 ^  0000
Verified with key: 00-> 0010 =  0011 ^  1000 ^  1001  Verified:  0010
Verified with key: 01-> 0100 =  0101 ^  1110 ^  1111  Verified:  0100
Verified with key: 10-> 1101 =  1100 ^  0111 ^  0110  Verified:  1101
Verified with key: 11-> 1011 =  1010 ^  0001 ^  0000  Verified:  1011

Verification:
Equation:               0111 =  0100 ^  0010 ^  0001
Verified with key: 00-> 0111 =  1100 ^  0011 ^  1000  Verified:  0111
Verified with key: 01-> 0001 =  1010 ^  0101 ^  1110  Verified:  0001
Verified with key: 10-> 1000 =  0011 ^  1100 ^  0111  Verified:  1000
Verified with key: 11-> 1110 =  0101 ^  1010 ^  0001  Verified:  1110

Verification:
Equation:               1011 =  1000 ^  0010 ^  0001
Verified with key: 00-> 1010 =  0001 ^  0011 ^  1000  Verified:  1010
Verified with key: 01-> 1100 =  0111 ^  0101 ^  1110  Verified:  1100
Verified with key: 10-> 0101 =  1110 ^  1100 ^  0111  Verified:  0101
Verified with key: 11-> 0011 =  1000 ^  1010 ^  0001  Verified:  0011

Verification:
Equation:               1101 =  1000 ^  0100 ^  0001
Verified with key: 00-> 0101 =  0001 ^  1100 ^  1000  Verified:  0101
Verified with key: 01-> 0011 =  0111 ^  1010 ^  1110  Verified:  0011
Verified with key: 10-> 1010 =  1110 ^  0011 ^  0111  Verified:  1010
Verified with key: 11-> 1100 =  1000 ^  0101 ^  0001  Verified:  1100

Verification:
Equation:               1110 =  1000 ^  0100 ^  0010
Verified with key: 00-> 1110 =  0001 ^  1100 ^  0011  Verified:  1110
Verified with key: 01-> 1000 =  0111 ^  1010 ^  0101  Verified:  1000
Verified with key: 10-> 0001 =  1110 ^  0011 ^  1100  Verified:  0001
Verified with key: 11-> 0111 =  1000 ^  0101 ^  1010  Verified:  0111

Verification:
Equation:               1111 =  1000 ^  0100 ^  0010 ^  0001 ^  0000
Verified with key: 00-> 1111 =  0001 ^  1100 ^  0011 ^  1000 ^  1001  Verified:  1111
Verified with key: 01-> 1001 =  0111 ^  1010 ^  0101 ^  1110 ^  1111  Verified:  1001
Verified with key: 10-> 0000 =  1110 ^  0011 ^  1100 ^  0111 ^  0110  Verified:  0000
Verified with key: 11-> 0110 =  1000 ^  0101 ^  1010 ^  0001 ^  0000  Verified:  0110
```

PART 4 (4.4 & 4.5)

How to compile the program:

---

1. Open your command prompt
2. Navigate to folder where java file is stored
3. javac MDES.java
4. java MDES

4) Encryption & Decryption output screenshot

```
[sallyyeo@sallys-MBP desktop % javac MDES.java
[sallyyeo@sallys-MBP desktop % java MDES
MDES Encryption:
Key\Message:      0000  0001  0010  0011  0100  0101  0110  0111  1000  1001  1010  1011  1100  1101  1110  1111
   00             0000  0100  1000  1100  0001  0101  1111  1011  0010  1001  1010  0111  0011  1110  1101  0110
   01             1111  0100  0001  1100  1000  0101  0000  1011  0010  0110  1010  1110  0011  0111  1101  1001
   10             0110  0010  1000  1100  0001  0101  1001  1101  0100  1111  1010  0111  0011  1110  1011  0000
   11             1001  0010  0001  1100  1000  0101  0110  1101  0100  0000  1010  1110  0011  0111  1011  1111

MDES Decryption:
Key\Message:      0000  0001  0010  0011  0100  0101  0110  0111  1000  1001  1010  1011  1100  1101  1110  1111
   00             0000  0100  1000  1100  0001  0101  1111  1011  0010  1001  1010  0111  0011  1110  1101  0110
   01             1111  0100  0001  1100  1000  0101  0000  1011  0010  0110  1010  1110  0011  0111  1101  1001
   10             0110  0010  1000  1100  0001  0101  1001  1101  0100  1111  1010  0111  0011  1110  1011  0000
   11             1001  0010  0001  1100  1000  0101  0110  1101  0100  0000  1010  1110  0011  0111  1011  1111
```

5) Unable to verify the equation due to non-linear relationship

Output Screenshot

```
Verification:
Equation:                  1100 =  1000 ^  0100 ^  0000
Verified with key: 00->    0011 =  0010 ^  0001 ^  0000  Verified:  0011
Verified with key: 01->    0011 =  0010 ^  1000 ^  1111
Verified with key: 10->    0011 =  0100 ^  0001 ^  0110  Verified:  0011
Verified with key: 11->    0011 =  0100 ^  1000 ^  1001
```

See next page for the rest of equations

PART 4 (4.4 & 4.5)

5) The rest of the equation output screenshot

```
Verification:
Equation:                0011 =  0010 ^  0001 ^  0000
Verified with key: 00->  1100 =  1000 ^  0100 ^  0000  Verified:  1100
Verified with key: 01->  1100 =  0001 ^  0100 ^  1111
Verified with key: 10->  1100 =  1000 ^  0010 ^  0110  Verified:  1100
Verified with key: 11->  1100 =  0001 ^  0010 ^  1001

Verification:
Equation:                1010 =  1000 ^  0010 ^  0000
Verified with key: 00->  1010 =  0010 ^  1000 ^  0000  Verified:  1010
Verified with key: 01->  1010 =  0010 ^  0001 ^  1111
Verified with key: 10->  1010 =  0100 ^  1000 ^  0110  Verified:  1010
Verified with key: 11->  1010 =  0100 ^  0001 ^  1001

Verification:
Equation:                1001 =  1000 ^  0001 ^  0000
Verified with key: 00->  1001 =  0010 ^  0100 ^  0000
Verified with key: 01->  0110 =  0010 ^  0100 ^  1111
Verified with key: 10->  1111 =  0100 ^  0010 ^  0110
Verified with key: 11->  0000 =  0100 ^  0010 ^  1001

Verification:
Equation:                0110 =  0100 ^  0010 ^  0000
Verified with key: 00->  1111 =  0001 ^  1000 ^  0000
Verified with key: 01->  0000 =  1000 ^  0001 ^  1111
Verified with key: 10->  1001 =  0001 ^  1000 ^  0110
Verified with key: 11->  0110 =  1000 ^  0001 ^  1001

Verification:
Equation:                0101 =  0100 ^  0001 ^  0000
Verified with key: 00->  0101 =  0001 ^  0100 ^  0000  Verified:  0101
Verified with key: 01->  0101 =  1000 ^  0100 ^  1111
Verified with key: 10->  0101 =  0001 ^  0010 ^  0110  Verified:  0101
Verified with key: 11->  0101 =  1000 ^  0010 ^  1001

Verification:
Equation:                0011 =  0010 ^  0001 ^  0000
Verified with key: 00->  1100 =  1000 ^  0100 ^  0000  Verified:  1100
Verified with key: 01->  1100 =  0001 ^  0100 ^  1111
Verified with key: 10->  1100 =  1000 ^  0010 ^  0110  Verified:  1100
Verified with key: 11->  1100 =  0001 ^  0010 ^  1001

Verification:
Equation:                0111 =  0100 ^  0010 ^  0001
Verified with key: 00->  1011 =  0001 ^  1000 ^  0100
Verified with key: 01->  1011 =  1000 ^  0001 ^  0100
Verified with key: 10->  1101 =  0001 ^  1000 ^  0010
Verified with key: 11->  1101 =  1000 ^  0001 ^  0010

Verification:
Equation:                1011 =  1000 ^  0010 ^  0001
Verified with key: 00->  0111 =  0010 ^  1000 ^  0100
Verified with key: 01->  1110 =  0010 ^  0001 ^  0100
Verified with key: 10->  0111 =  0100 ^  1000 ^  0010
Verified with key: 11->  1110 =  0100 ^  0001 ^  0010

Verification:
Equation:                1101 =  1000 ^  0100 ^  0001
Verified with key: 00->  1110 =  0010 ^  0001 ^  0100
Verified with key: 01->  0111 =  0010 ^  1000 ^  0100
Verified with key: 10->  1110 =  0100 ^  0001 ^  0010
Verified with key: 11->  0111 =  0100 ^  1000 ^  0010

Verification:
Equation:                1110 =  1000 ^  0100 ^  0010
Verified with key: 00->  1101 =  0010 ^  0001 ^  1000
Verified with key: 01->  1101 =  0010 ^  1000 ^  0001
Verified with key: 10->  1011 =  0100 ^  0001 ^  1000
Verified with key: 11->  1011 =  0100 ^  1000 ^  0001

Verification:
Equation:                1111 =  1000 ^  0100 ^  0010 ^  0001 ^  0000
Verified with key: 00->  0110 =  0010 ^  0001 ^  1000 ^  0100 ^  0000
Verified with key: 01->  1001 =  0010 ^  1000 ^  0001 ^  0100 ^  1111
Verified with key: 10->  0000 =  0100 ^  0001 ^  1000 ^  0010 ^  0110
Verified with key: 11->  1111 =  0100 ^  1000 ^  0001 ^  0010 ^  1001
```

PART 4.6

How to compile the program:

---

1. Open your command prompt
2. Navigate to folder where java file is stored
3. javac MDES_ECB_CBC.java
4. For ECB mode please key following to the terminal
   java MDES_ECB_CBC -key <2-bit binary string> -mode ECB <-encrypt or -decrypt> <7-char Hex String>
   Example: java MDES_ECB_CBC -key 10 -mode ECB -encrypt f4a5a32
   Example: java MDES_ECB_CBC -key 01 -mode ECB -decrypt b6f7a11
5. For CBC mode please key the following format to the terminal
   java MDES_ECB_CBC -key <2-bit binary string> -mode CBB -iv <a hex char> <-encrypt or -decrypt> <7-char Hex String>
   Example: java MDES_ECB_CBC -key 11 -mode CBC -iv a -encrypt 2a45def
   Example: java MDES_ECB_CBC -key 00 -mode CBC -iv 4 -decrypt b412ab2

CBC Mode Output Screenshot

```
sallyyeo@sallys-MBP desktop % java MDES_ECB_CBC -key 11 -mode CBC -iv a -encrypt 2a45def
CBC Encryption Mode
------------------------------------------------------------------
Input Text                    : 2a45def
CBC Encrypted Output          : d490d96
sallyyeo@sallys-MBP desktop % java MDES_ECB_CBC -key 00 -mode CBC -iv 4 -decrypt b412ab2
CBC Decryption Mode
------------------------------------------------------------------
Input Text                    : b412ab2
CBC decrypted Output          : f6097ef
```

ECB Mode Output Screenshot

```
sallyyeo@sallys-MBP desktop % java MDES_ECB_CBC -key 10 -mode ECB -encrypt f4a5a32
ECB Encryption Mode
------------------------------------------------------------------
Input Text                    : f4a5a32
ECB Encrypted Output          : 01555c8
------------------------------------------------------------------

sallyyeo@sallys-MBP desktop % java MDES_ECB_CBC -key 01 -mode ECB -decrypt b6f7a11
ECB Decryption Mode
------------------------------------------------------------------
Input Text                    : b6f7a11
ECB decrypted Output          : b09b544
------------------------------------------------------------------
```

PART 5

How to compile the program:

---

1. Open your command prompt
2. Navigate to folder where files are stored
3. javac PartFive.java
4. java PartFive

Section 2: 4-bit CFB Tea to encrypt student number 4603229

CFB Output Screenshot

```
[sallyyeo@sallys-MBP desktop % javac PartFive.java
[sallyyeo@sallys-MBP desktop % java PartFive
 4 Bit CFB TEA Encryption
 ----------------------------------------------------------------
 k-Bit Size                    : 4
 Input Decimal                 : 4603229
 Input Binary                  : 10001100011110101011101
 input Hex                     : 463d5d
 inputH length                 : 6
 inputTextBlock                : 6

 CFB Block 0
 IV                            : 11111111111111111111111111111111111111111111111111111111111111111111
 TEA output                    : 10111101110000010101111110010000011010000110000000011001000010
 Most left 4-bit of TEA output : 1011
 Plaintext                     : 10001100011110101011101
 Most left 4-bit Plaintext     : 100
 TEA output XOR k-bit Input    : 1111
 Cipher Text                   : 1111

 CFB Block 1
 IV                            : 11111111111111111111111111111111111111111111111111111111111111111111
 TEA output                    : 10111101110000010101111110010000011010000110000000011001000010
 Most left 4-bit of TEA output : 1011
 Plaintext                     : 1100011110101011101
 Most left 4-bit Plaintext     : 110
 TEA output XOR k-bit Input    : 1101
 Cipher Text                   : 11111101

 CFB Block 2
 IV                            : 11111111111111111111111111111111111111111111111111111111111111111101
 TEA output                    : 11111110100000110110011101111110110100100001000001111001111110
 Most left 4-bit of TEA output : 111
 Plaintext                     : 11110101011101
 Most left 4-bit Plaintext     : 11
 TEA output XOR k-bit Input    : 100
 Cipher Text                   : 111111010100

 CFB Block 3
 IV                            : 11111111111111111111111111111111111111111111111111111111111111010100
 TEA output                    : 11010010110110011100111100010010011011000100001010100011111001100
 Most left 4-bit of TEA output : 1101
 Plaintext                     : 110101011101
 Most left 4-bit Plaintext     : 1101
 TEA output XOR k-bit Input    : 0
 Cipher Text                   : 1111110101000000

 CFB Block 4
 IV                            : 11111111111111111111111111111111111111111111111111111111111110101000000
 TEA output                    : 11000011100000100000111010011001001011000011010001010110010100000
 Most left 4-bit of TEA output : 1100
 Plaintext                     : 1011101
 Most left 4-bit Plaintext     : 101
 TEA output XOR k-bit Input    : 1001
 Cipher Text                   : 11111101010000001001

 CFB Block 5
 IV                            : 11111111111111111111111111111111111111111111111111111101010000001001
 TEA output                    : 10110111011101011001010010011000100101111011001001110110111000 0
 Most left 4-bit of TEA output : 101
 Plaintext                     : 1101
 Most left 4-bit Plaintext     : 1101
 TEA output XOR k-bit Input    : 1000
 Cipher Text                   : 11111101010000001011000

 Encrypted Student UOW ID 4603229 using 4-bits CFB TEA algorithm: fd4098
 Duration for encrypting 4-bits CFB with TEA encryption: 1.2981769E7 nanoseconds
```

PART 5

Section 3: c-bit OFB Tea to encrypt student number 4603229

OFB Output Screenshot

```
2 Bit OFB TEA Encryption
----------------------------------------------------------------
k-Bit Size                    : 2
Input Decimal                 : 4603229
Input Binary                  : 10001100011110101011101
Input Hex                     : 463d5d
Input Hex length              : 6
Input Blocks                  : 12

OFB Block 0
IV                            : 11111111111111111111111111111111111111111111111111111111111111111111
TEA output                    : 10111101110000010101111110010000001101000011000000000011001000010
Most left k-bit of TEA output : 10
Plaintext                     : 10001100011110101011101
Most left k-bit of Input      : 1
TEA output XOR k-bit Input    : 11
Cipher Text                   : 11

OFB Block 1
IV                            : 11111111111111111111111111111111111111111111111111111111111111111110
TEA output                    : 10000101100000110101011111000101110111011001001100111110001101011
Most left k-bit of TEA output : 10
Plaintext                     : 1100011110101011101
Most left k-bit of Input      : 0
TEA output XOR k-bit Input    : 10
Cipher Text                   : 1110

OFB Block 2
IV                            : 11111111111111111111111111111111111111111111111111111111111111111010
TEA output                    : 11111100001011010010110010101010011101101010010010111010011101
Most left k-bit of TEA output : 0
Plaintext                     : 1100011110101011101
Most left k-bit of Input      : 1
TEA output XOR k-bit Input    : 1
Cipher Text                   : 111001

OFB Block 3
IV                            : 11111111111111111111111111111111111111111111111111111111111111101000
TEA output                    : 11101101001001000100001111111001010110101000111101000110001000 1
Most left k-bit of TEA output : 1
Plaintext                     : 100011110101011101
Most left k-bit of Input      : 10
TEA output XOR k-bit Input    : 11
Cipher Text                   : 11100111

OFB Block 4
IV                            : 11111111111111111111111111111111111111111111111111111111111110100001
TEA output                    : 10001101001100100111011010011101010011111100010111100111111110111
Most left k-bit of TEA output : 10
Plaintext                     : 11110101011101
Most left k-bit of Input      : 0
TEA output XOR k-bit Input    : 10
Cipher Text                   : 1110011110

OFB Block 5
IV                            : 11111111111111111111111111111111111111111111111111111111111010000110
TEA output                    : 11110100111110110101000011101111010001100001111001110001010110
Most left k-bit of TEA output : 1
Plaintext                     : 11110101011101
Most left k-bit of Input      : 11
TEA output XOR k-bit Input    : 10
Cipher Text                   : 111001111010

OFB Block 6
IV                            : 11111111111111111111111111111111111111111111111111111111101000011001
TEA output                    : 10001111001010101101100010101110101011100011101110001011011000 10
Most left k-bit of TEA output : 10
Plaintext                     : 110101011101
Most left k-bit of Input      : 11
TEA output XOR k-bit Input    : 1
Cipher Text                   : 11100111101001
```

PART 5

Section 3: OFB Output Screenshot

```
OFB Block 7
IV                            : 11111111111111111111111111111111111111111111111111111111100001100110
TEA output                    : 11110001010101110110001101000001011010000011100110000011111 1010
Most left k-bit of TEA output : 1
Plaintext                     : 101011101
Most left k-bit of Input      : 1
TEA output XOR k-bit Input    : 0
Cipher Text                   : 1110011110100100

OFB Block 8
IV                            : 11111111111111111111111111111111111111111111111111111010000110011001
TEA output                    : 11110011010011111100110111011110100011010001101110001100011110 0
Most left k-bit of TEA output : 1
Plaintext                     : 1011101
Most left k-bit of Input      : 1
TEA output XOR k-bit Input    : 0
Cipher Text                   : 111001111010010000

OFB Block 9
IV                            : 11111111111111111111111111111111111111111111111111101000011001100101
TEA output                    : 10000111101101010110001110110101010010111011101010011010111 01
Most left k-bit of TEA output : 0
Plaintext                     : 11101
Most left k-bit of Input      : 1
TEA output XOR k-bit Input    : 1
Cipher Text                   : 11100111101001000001

OFB Block 10
IV                            : 11111111111111111111111111111111111111111111111101000011001100100100
TEA output                    : 11010010110001110111101011010000110101010110011010011011001 0101
Most left k-bit of TEA output : 11
Plaintext                     : 1101
Most left k-bit of Input      : 11
TEA output XOR k-bit Input    : 0
Cipher Text                   : 1110011110100100000100

OFB Block 11
IV                            : 11111111111111111111111111111111111111111111111010000110011001010011
TEA output                    : 11101110010001011111010000111100100011100101110010111101100110 11
Most left k-bit of TEA output : 11
Plaintext                     : 1
Most left k-bit of Input      : 1
TEA output XOR k-bit Input    : 10
Cipher Text                   : 11100111101001000001 0010

Encrypted Student UOW ID 4603229 using 2-bits OFB TEA algorithm: e7a412
Duration for encrypting 2-bits OFB with TEA encryption: 8303769 nanoseconds
```

Section 3: Compare timing of 4-bit CFB and 2-bit OFB

```
Encrypted Student UOW ID 4603229 using 4-bits CFB TEA algorithm: fd4098
Duration for encrypting 4-bits CFB with TEA encryption: 1.2981769E7 nanoseconds
```

```
Encrypted Student UOW ID 4603229 using 2-bits OFB TEA algorithm: e7a412
Duration for encrypting 2-bits OFB with TEA encryption: 8303769 nanoseconds
```

The timing for 4-bit CFB is longer than timing of 2-bit OFB because CFB mode must wait for ciphertext output to produce the next round IV. While for OFB the IV can be pre prepared by XOR the TEA output with the previous round IV.

PART 6

How to compile the program:

---

1. Open your command prompt
2. Navigate to folder where files are stored
3. javac streamCipher.java
4. java streamCipher

6a. Decryption is done via Mi = Ci – Ki (mod 26)

Firstly, we Map A-Z to {0, 1, 2, …, 25}
Then we calculate the keys of each character.
Apply the formula Mi = Ci – Ki (mod 26), we get the plaintext back.

6b output

```
[sallyyeo@sallys-MacBook-Pro desktop % javac streamCipher.java
[sallyyeo@sallys-MacBook-Pro desktop % java streamCipher
Enter 1 to Encrypt or 2 to Decrypt:
1
Enter the message and key:
wollongong 3
The input text: WOLLONGONG

The encrypted text is:GMSMUDJFZE
[sallyyeo@sallys-MacBook-Pro desktop % java streamCipher
Enter 1 to Encrypt or 2 to Decrypt:
2
Enter the cypher text and key:
GMSMUDJFZE 3
The input text: GMSMUDJFZE

The decrypted text is:WOLLONGONG
```

6c. WOLLONGONG is encrypted to GMSMUDJFZE using key = 3.

6d. MQJJ is decrypted to CSCI using key = 3.

```
[sallyyeo@sallys-MacBook-Pro desktop % java streamCipher
Enter 1 to Encrypt or 2 to Decrypt:
2
Enter the cypher text and key:
mqjj 3
The input text: MQJJ

The decrypted text is:CSCI
```

Reference List

Part 1 Affine Cipher
Line 120-215 of AffineCipher.java
Ref: https://www.geeksforgeeks.org/implementation-affine-cipher/
Ref: https://stackoverflow.com/questions/19605465/how-to-write-logic-for-affine-cipher-decryption-in-java

Part 2 Decrypt Text using Affine Cipher
Line 59-85 of AffineCipher.java
Ref: https://stackoverflow.com/questions/21750365/how-to-find-the-most-frequently-occurring-character-in-a-string-with-java

Line 92-117 of AffineCipher.java
Ref: https://www.w3resource.com/java-exercises/string/java-string-exercise-34.php

Line 216-291 of AffineCipher.java
Ref: http://www.cs.trincoll.edu/~crypto/student/emilio/IC.java

Line 293-323 of AffineCipher.java
Ref: https://www.dreamincode.net/forums/topic/238757-index-of-coincidence/

Part 5 Modes of Block Cipher
Ref: Tutorial Slide KBitOFBExample.pdf