

2.1)

Task 1.Frequency Analysis and Decoding the ciphertext

I have taken the encrypted text present in the cipher text and then have performed the frequency analysis on the same and below are the count of each letter

| Code | Count |
|------|-------|
| P | 116 |
| O | 93 |
| L | 85 |
| A | 81 |
| Z | 79 |
| Y | 68 |
| F | 64 |
| W | 61 |
| H | 45 |

| | |
|---|----|
| S | 41 |
| X | 35 |
| K | 31 |
| G | 30 |
| C | 26 |
| V | 26 |
| N | 18 |
| J | 16 |
| U | 15 |
| Q | 12 |
| D | 7 |
| E | 6 |
| I | 4 |
| M | 1 |
| B | 1 |

The key is : 'SJMVBOLCKYUNQPTEWXDZFGRHIA' and the text is decoded as shown below

```
Terminator /bin/bash /bin/bash 80x24
xp aphwpo aploplhp ya vffs bfe vkja[09/20/22]seed@VM:~/lab02$ tr 'a-z' SJMVBOLC
KYUNQPTEWXDZFGRHIA <CipherText.txt >Out.txt
bash: CipherTextSJMVBOLCKYUNQPTEWXDZFGRHIA: No such file or directory
[09/20/22]seed@VM:~/lab02$ tr 'a-z' SJMVBOLCKYUNQPTEWXDZFGRHIA <CipherText.txt >
Out.txt
[09/20/22]seed@VM:~/lab02$ cat out.txt
cat: out.txt: No such file or directory
[09/20/22]seed@VM:~/lab02$ cat Out.txt
COMPUTER SCIENCE IS RAPIDLY CHANGING THE WORLD WITH NEW DEVELOPMENTS HAPPENING E
VERY DAY A RIGOROUS EDUCATION COMBINING THE THEORY OF INFORMATION AND COMPUTATIO
N WITH HANDSON SYSTEMS AND SOFTWARE DESIGN IS THE KEY TO SUCCESS AS ONE OF THE O
LDEST COMPUTER SCIENCE DEPARTMENTS IN THE CHICAGO AREA THE CS DEPARTMENT AT IIT
HAS A LONG HISTORY OF MEETING THIS CHALLENGE THROUGH QUALITY EDUCATION IN SMALL
CLASSROOM ENVIRONMENTS ALONG WITH INTERNSHIP AND RESEARCH OPPORTUNITIES IN INDUS
TRY AND NATIONAL LABORATORIES
IIT STUDENTS WORK WITH OUR FACULTY ON WORLDCLASS RESEARCH IN AREAS THAT INCLUDE
DATA SCIENCE DISTRIBUTED SYSTEMS INFORMATION RETRIEVAL COMPUTER NETWORKING INTEL
LIGENT INFORMATION SYSTEMS AND ALGORITHMS
THE DEPARTMENT OFFERS BACHELOR OF SCIENCE MASTER OF SCIENCE PROFESSIONAL MASTER
AND PHD DEGREES PLUS GRADUATE CERTIFICATES ACCELERATED COURSES AND NONDEGREE STU
DY PARTTIME STUDENTS CAN TAKE EVENING CLASSES AND LONGDISTANCE STUDENTS CAN EARN
MASTERS DEGREES ONLINE STUDENTS RATE OUR TEACHING AS AMONG THE BEST AT THE UNIV
ERSITY AND OUR FACULTY HAVE WON NUMEROUS TEACHING AWARDS
THE SECRET SENTENCE IS GOOD JOB GUYS[09/20/22]seed@VM:~/lab02$
```

2.2).

Task 2). Encryption using different ciphers and modes

I have considered the plain text ‘Introduction to Information Security course is an amazing course’ and stored it in a file ‘plain.txt’ and then made use of different encryption methods to encrypt the plain.txt file.

Firstly, I went through the different modes of operation in block cipher:

There are totally 5 different modes of operation in block cipher:

- 1).ECB(Electronic Codebook Mode)
- 2).CBC(Cipher Block Chaining Mode)
- 3). CFB(Cipher Feedback Mode)
- 4). OFB(Output Feedback Mode)
- 5). CTR(Counter Mode)

I am making use of CFB,OFB and CTR modes.

CFB- The CFB mode uses block cipher but in a way acts as Stream cipher. I mean data is encrypted in smaller 8 bits rather than a predefined size of 64 bits. In this mode, 64 bits of initialization vector is used which is kept in 64 bits of shift register. The cipher text of the one block is given as input to the shift register of the next block.

OFB- The OFB mode is similar in structure to CFB but here the cipher text is not fed to the next block instead the output of the encryption of 1st is fed to the next block. The biggest advantage of using this mode is that even if there is a small error in individual bits then it will not corrupt the whole encrypted message.

CTR- The CTR mode is similar to OFB mode with slight changes. Instead of Nonce(as in OFB), it makes use of Counters as input to the algorithm and there is no chaining process as in CFB and OFB.

AES Cipher:

```
[09/17/22]seed@VM:~/lab2$ openssl enc -aes-128-cfb -e -in plain.txt -out cipher.bin \-K 00010203040506070809aabbccddeff \-iv 0a0b0c0d0e0f010203040506070809
[09/17/22]seed@VM:~/lab2$ xxd cipher.bin
00000000: 8cd5 fd87 43e8 327c 2a70 2a42 31bb e9ca ....C.2|*p*B1...
00000010: a6a6 9dcc 69aa 2734 fedb d8e5 f2e9 efac ....i.'4.....
00000020: 63b4 0452 b0df 4a5b 21a6 dba3 5a51 95d3 c..R..J[!...ZQ...
00000030: a22a 9d38 6170 c938 d9f0 .*..8ap.8..
[09/17/22]seed@VM:~/lab2$ openssl enc -aes-128-ofb -e -in plain.txt -out cipher1.bin \-K 00010203040506070809aabbccddeff \-iv 0a0b0c0d0e0f010203040506070809
[09/17/22]seed@VM:~/lab2$ xxd cipher1.bin
00000000: 8cd5 fd87 43e8 327c 2a70 2a42 31bb e9ca ....C.2|*p*B1...
00000010: 89f0 467a 2480 e578 dec9 3c4f c418 e93d ..Fz$.X.<0...=
00000020: f5a4 537e 2cca 6556 9393 4c05 6d44 552d ..S.,.eV..L.mDU-
00000030: e423 8ca4 36ff 1a57 a6cd .#.6..W..
[09/17/22]seed@VM:~/lab2$ openssl enc -aes-128-ctr -e -in plain.txt -out cipher1.bin \-K 00010203040506070809aabbccddeff \-iv 0a0b0c0d0e0f010203040506070809
[09/17/22]seed@VM:~/lab2$ xxd cipher1.bin
00000000: 8cd5 fd87 43e8 327c 2a70 2a42 31bb e9ca ....C.2|*p*B1...
00000010: c1f1 774d bc12 b4bc 0b61 1895 18b2 21cf ..WM.....!
00000020: 20c3 2068 5c8a 67ff 87a9 2467 34f5 3885 .h\,g...$g4.8.
00000030: b51e 7a88 46cf 5961 f0c5 ..Z.F.Ya..
```

Blowfish Cipher:

```
cipher1.bin cipher.bin cipher1.txt Key plain.txt
[09/17/22]seed@M:~/Lab2$ openssl enc -bf-cfb -e -in plain.txt -out cipher2.bin \-K 00010203040506070809aabbccddeff \-iv 0a0b0c0d0e0f010203040506070809
[09/17/22]seed@M:~/Lab2$ xxd cipher2.bin
00000000: ea17 8bf1 cf59 0bdb 79c4 8240 7df0 943f ....Y.y..@..?
00000010: a8cd 68c5 e611 5e2d 86f5 ac64 0bd6 7380 ..h..^....d.s.
00000020: 9e0a 8f9d b8a7 b426 7aaf b198 d970 a537 .....&z....p.7
00000030: dce2 f8fc cc97 a82e e216 .....
[09/17/22]seed@M:~/Lab2$ openssl enc -bf-ofb -e -in plain.txt -out cipher3.bin \-K 00010203040506070809aabbccddeff \-iv 0a0b0c0d0e0f010203040506070809
[09/17/22]seed@M:~/Lab2$ xxd cipher3.bin
00000000: ea17 8bf1 cf59 0bdb 6c8b file 3717 38d9 ....Y.l...7.8.
00000010: e3e2 09bc c7e3 00fc 90f9 b8b5 f17e 6db7 .....-m.
```

Note: Blowfish cipher does not support Counter mode.

ARC2 Cipher:

```
[09/17/22]seed@M:~/lab2$ openssl enc -rc2-cfb -e -in plain.txt -out cipher4.bin \-K 00010203040506070809aabbccddeff \-iv 0a0b0c0d0e0f010203040506070809
[09/17/22]seed@M:~/lab2$ xxd cipher4.bin
00000000: 4073 f1cb a7c6 99ba 7726 3d1a 3d84 1f71 @s.....w$=.=..q
00000010: c84a 492f eb28 73ba c044 331b 5bd1 917b .JI/(s..D3.[{.
00000020: 41e0 56f7 fc8f 2232 f260 744b e2dc 8988 A.V...2..tK....
00000030: 3309 ca92 2d81 5bab 94b0 3....[...
[09/17/22]seed@M:~/Lab2$ openssl enc -rc2-ofb -e -in plain.txt -out cipher5.bin \-K 00010203040506070809aabbccddeff \-iv 0a0b0c0d0e0f010203040506070809
[09/17/22]seed@M:~/Lab2$ xxd cipher5.bin
00000000: 4073 f1cb a7c6 99ba adbb e095 ec92 adf6 @s.....w$=.=..q
00000010: 6158 06c6 2ed3 5f72 e243 290b 9e37 8aa8 aX...W..C)..7..
00000020: 858f 3164 26d8 d91c 58e0 7b5c 86ce 8637 ..1d&..X.{...7
00000030: 5b4f e0d9 d50e 2aa8 30e7 [0....*..0.
[09/17/22]seed@M:~/Lab2$ openssl enc -rc2-ecb -e -in plain.txt -out cipher5.bin \-K 00010203040506070809aabbccddeff \-iv 0a0b0c0d0e0f010203040506070809
warning: iv not use by this cipher
[09/17/22]seed@M:~/Lab2$ xxd cipher5.bin
00000000: 58af 5973 9b79 83e9 369f c0de eadb 4cdc X.Ys.y..6....L.
00000010: 5fa2 c424 9750 0a75 168e 656a 33fb 21d7 ..$.P.u..e'3.!
00000020: ad65 956a 2e83 d99a 6abb 4f92 706e 44d7 .e.j....j.0.bnD.
00000030: 337a d422 6e5a a4c8 89e8 5b2b 61cf be39 3z."nZ....[+a..9
[09/17/22]seed@M:~/lab2$
```

I have made use of xxd command to observe the encrypted contents in hexadecimal format.
2.3).

Task 3 : Encryption Mode- ECB vs CBC

I downloaded the pic_original.bmp file and placed it in the shared folder and then took the file from the shared folder and placed it in the lab2 folder in VM.

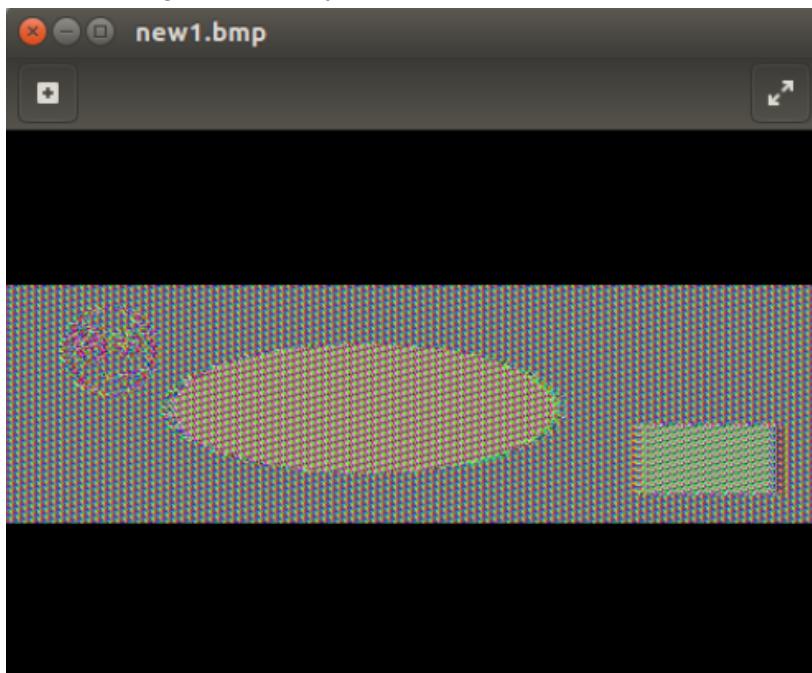
First I am encrypting the image pic_original.bmp using **AES-256 cipher in ECB mode** and the output file is new.bmp

```
[09/17/22]seed@VM:~/lab2$ ls  
cipher1.bin cipher2.bin cipher3.bin cipher4.bin cipher5.bin cipher.bin ciphertext.txt Key pic_original.bmp plain.txt  
[09/17/22]seed@VM:~/lab2$ openssl enc -aes-256-ecb -e -in pic_original.bmp -out new.bmp -K 125687  
[09/17/22]seed@VM:~/lab2$
```

As mentioned in the question, I copied the header from pic_original.bmp and tail from new.bmp and put both into the new bmp file i.e. new1.bmp.

```
[09/17/22]seed@VM:~/lab2$ head -c 54 pic_original.bmp >header  
[09/17/22]seed@VM:~/lab2$ tail -c +55 new.bmp >body  
[09/17/22]seed@VM:~/lab2$ cat header body > new1.bmp  
[09/17/22]seed@VM:~/lab2$ ls  
body cipher2.bin cipher4.bin cipher.bin header new1.bmp pic_original.bmp  
cipher1.bin cipher3.bin cipher5.bin ciphertext.txt Key new.bmp plain.txt  
[09/17/22]seed@VM:~/lab2$
```

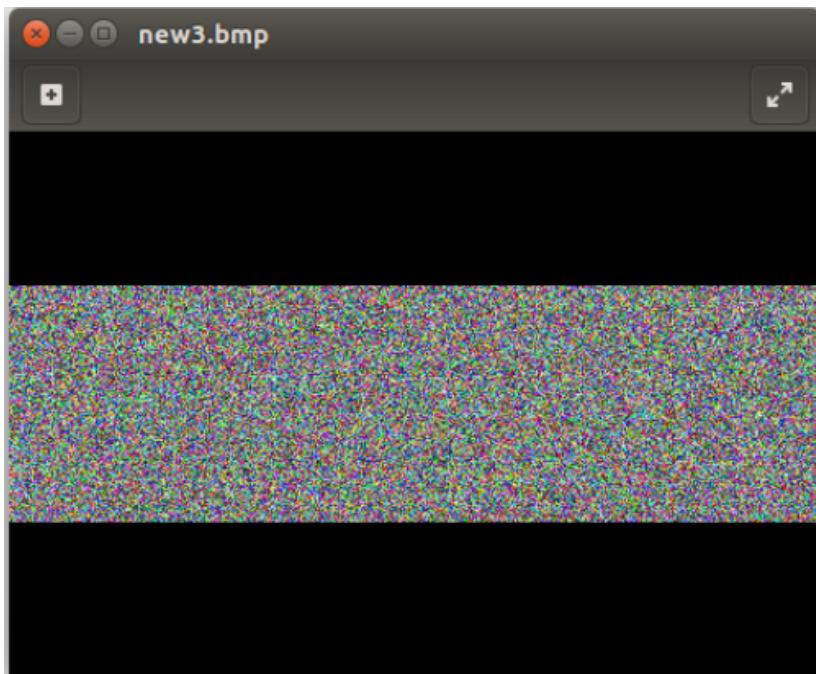
And the Image after encryption looks like as shown below:



The same process is repeated with the AES cipher in CBC mode.

```
[09/17/22]seed@VM:~/lab2$ openssl enc -aes-256-cbc -e -in pic_original.bmp -out new2.bmp -K 125687 -iv 0a0b0c0d0e0f010203040506070809  
[09/17/22]seed@VM:~/Lab2$ head -c 54 pic_original.bmp >header  
[09/17/22]seed@VM:~/Lab2$ tail -c +55 new2.bmp >body  
[09/17/22]seed@VM:~/Lab2$ cat header body > new3.bmp  
[09/17/22]seed@VM:~/lab2$
```

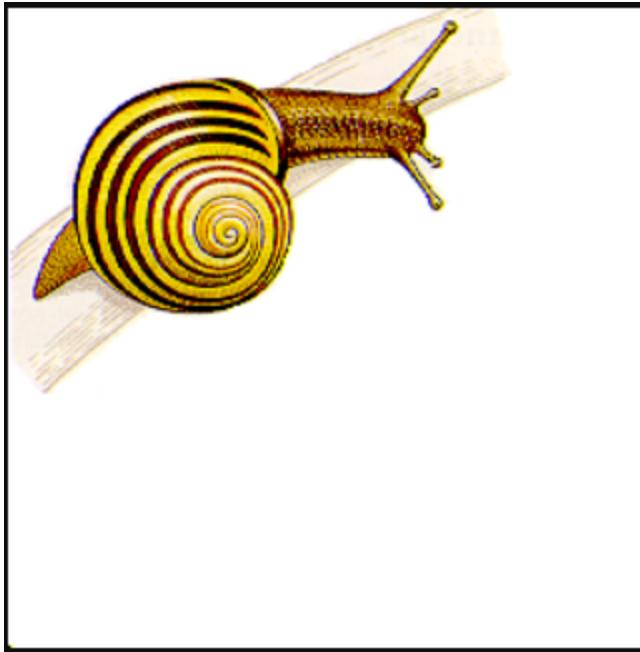
And the encrypted image is shown as below:



From both the encrypted pictures we can see that the image encrypted in ECB mode is very basic as we can roughly see the content i.e. the content of the image is not fully encrypted whereas the image encrypted in CBC mode is very good as it does not give any hint about the original image's content. I think in CBC mode, each cipher block is dependent on all the plaintext processed until that point and this adds an extra level of security to the encrypted data making it more complex.

2.3.1).

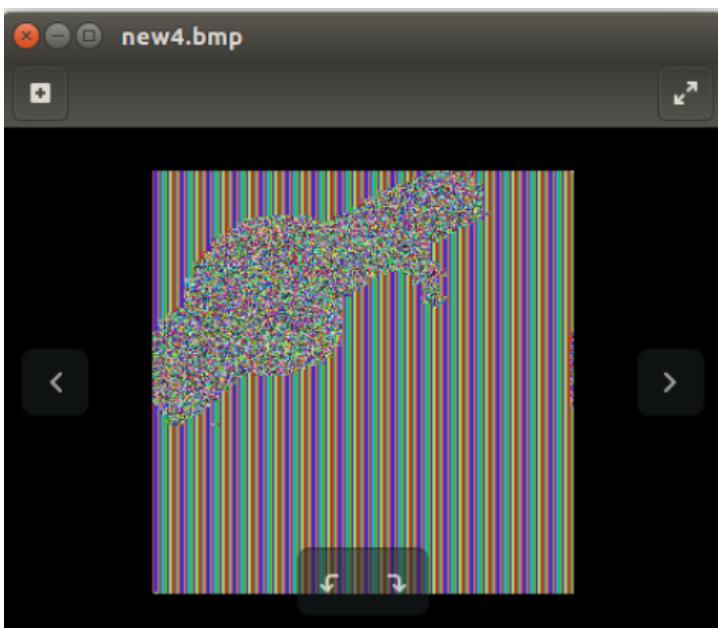
I choose the picture snail.bmp as shown below to experiment:



(I have pasted the screenshot of the image as .bmp is image is not supported in google docs)
Encrypting using in ecb mode:

```
[09/17/22]seed@VM:~/lab2$ openssl enc -aes-256-ecb -e -in snail.bmp -out new_snail.bmp -K 125687  
[09/17/22]seed@VM:~/lab2$ head -c 54 snail.bmp >header  
[09/17/22]seed@VM:~/lab2$ tail -c +55 new_snail.bmp >body  
[09/17/22]seed@VM:~/lab2$ cat header body > new4.bmp  
[09/17/22]seed@VM:~/lab2$
```

And the resulting encrypted image is



Encryption in cbc mode :

```
[09/17/22]seed@VM:~/lab2$ openssl enc -aes-256-cbc -e -in snail.bmp -out new_snail_1.bmp -K 125687 -iv 0a0b0c0d0e0f010203040506070809  
[09/17/22]seed@VM:~/lab2$ head -c 54 snail.bmp >header  
[09/17/22]seed@VM:~/lab2$ tail -c +55 new_snail_1.bmp >body  
[09/17/22]seed@VM:~/lab2$ cat header body > new6.bmp  
[09/17/22]seed@VM:~/lab2$
```



2.4).

Task 4 Padding:

In this task as per the question I first created three files of size 5bytes(f1.txt), 10 bytes(f2.txt) and 16 bytes(f3.txt) as shown below:

```
[09/18/22]seed@VM:~/.../padding$ ls  
[09/18/22]seed@VM:~/.../padding$ echo -n "12345" > f1.txt  
[09/18/22]seed@VM:~/.../padding$ echo -n "1234512345" > f2.txt  
[09/18/22]seed@VM:~/.../padding$ echo -n "1234512345123456" > f3.txt  
[09/18/22]seed@VM:~/.../padding$ ls  
f1.txt  f2.txt  f3.txt  
[09/18/22]seed@VM:~/.../padding$
```

1). Now I am making use of the f1.txt(5 bytes) file to encrypt using aes-128 cipher in different modes ECB,CBC,CFB and OFB.

And the Key I am using to encrypt is : 77217A25432A462D4A614E645267556B and IV

In ECB mode, there was padding :

```
/bin/bash
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-ecb -e -in f1.txt -out cipher_ecb.bin -K 77217A25432A462D4A614E645267556B
[09/18/22]seed@VM:~/.../padding$ ls -l f1.txt cipher_ecb.bin
-rw-rw-r-- 1 seed seed 16 Sep 18 19:13 cipher_ecb.bin
-rw-rw-r-- 1 seed seed 5 Sep 18 18:19 f1.txt
[09/18/22]seed@VM:~/.../padding$
```

In CBC mode, there was padding:

```
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cbc -e -in f1.txt -out cipher_cbc.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ ls -l f1.txt cipher_cbc.bin
-rw-rw-r-- 1 seed seed 16 Sep 18 19:16 cipher_cbc.bin
-rw-rw-r-- 1 seed seed 5 Sep 18 18:19 f1.txt
[09/18/22]seed@VM:~/.../padding$
```

In CFB mode, there was **no padding**:

```
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cfb -e -in f1.txt -out cipher_cfb.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ ls -l f1.txt cipher_cfb.bin
-rw-rw-r-- 1 seed seed 5 Sep 18 19:17 cipher_cfb.bin
-rw-rw-r-- 1 seed seed 5 Sep 18 18:19 f1.txt
[09/18/22]seed@VM:~/.../padding$
```

In OFB mode, there was **no padding**:

```
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-ofb -e -in f1.txt -out cipher_ofb.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ ls -l f1.txt cipher_ofb.bin
-rw-rw-r-- 1 seed seed 5 Sep 18 19:19 cipher_ofb.bin
-rw-rw-r-- 1 seed seed 5 Sep 18 18:19 f1.txt
[09/18/22]seed@VM:~/.../padding$
```

The ECB and CBC mode needs to have a plain text size in the multiple of 128 bits i.e. 16 bytes and in our case we only have 5 bytes of plain text. ECB and CBC dont have mechanisms to encrypt if the size of the plain text is less than 16 bytes so padding is done and that's the reason we are finding the size of encrypted file as 16 bytes.

Whereas in OFB and CFB mode they have the mechanisms which makes us feel block ciphers as stream ciphers and these modes where CFB has identical encryption and decryption functions and as a result doesn't require padding to the plain text.

2). Encrypting and Decrypting 5 bytes(f1.txt) file

Encrypting the f1.txt :

```
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cbc -e -in f1.txt -out cipher_cbc_f1.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ ls
cipher_cbc.bin      cipher_cfb.bin   cipher_ofb.bin  f2.txt
cipher_cbc_f1.bin    cipher_ecb.bin   f1.txt        f3.txt
```

Decrypting the file without removing padding and we can see that 11 bytes have been padded zero since we had only 5 bytes of plain text as shown below:

```
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cbc -d -nopad -in cipher_cbc_f1.bin -out f1_nopad.txt -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ hexdump -C f1_nopad.txt
00000000  31 32 33 34 35 0b |12345.....|
00000010
[09/18/22]seed@VM:~/.../padding$
```

Encrypting and Decrypting 10 bytes file:

Here 6 bytes have been padded as zero because we have plain text of 10 bytes only.

```
/bin/bash
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cbc -e -in f2.txt -out cipher_cbc_f2.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cbc -d -nopad -in cipher_cbc_f2.bin -out f2_nopad.txt -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ hexdump -C f2_nopad.txt
00000000  31 32 33 34 35 31 32 33 34 35 06 06 06 06 06 06 |1234512345.....|
00000010
[09/18/22]seed@VM:~/.../padding$
```

Encrypting and Decrypting 16 bytes file:

Here there is no padding as we have a plaintext of 16 bytes which is the size of one block.

```
/bin/bash
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cbc -e -in f3.txt -out cipher_cbc_f3.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cbc -d -nopad -in cipher_cbc_f3.bin -out f3_nopad.txt -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/18/22]seed@VM:~/.../padding$ hexdump -C f3_nopad.txt
00000000  31 32 33 34 35 31 32 33 34 35 31 32 33 34 35 36 |1234512345123456|
00000010  10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 |.....|
00000020
[09/18/22]seed@VM:~/.../padding$ xxd f3_nopad.txt
00000000: 3132 3334 3531 3233 3435 3132 3334 3536  1234512345123456
00000010: 1010 1010 1010 1010 1010 1010 1010 1010 .....
```

2.5)

Task 5: Error Propagation- Corrupted Cipher Text

ECB mode- Each block is encrypted and decrypted separately. So, I feel only a specific block i.e. the block which 55th bit blocks will get affected.

CBC mode- The ciphertext block is used for two things

- 1). As an input to the block cipher to calculate the output which is then XORed with the IV key to get the plain text.
- 2). As the input to the XOR to calculate the next plaintext block with the next decrypted block. So even in this scenario the bits which are changed or corrupted will affect the bit or probably only that block max.

CFB mode- I feel the result of CFB decryption will be similar to CBC.

OFB mode- Here the feedback is only in the key generation system. If the single digit is corrupted from a 1000 byte encrypted file then in plain text only that byte will be corrupted.

ECB mode:

Created 1000.txt (1000 bytes) file and then encrypted it as shown below:

```
/bin/bash
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-ecb -e -in 1000.txt -out cipher_1000.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
warning: iv not use by this cipher
[09/18/22]seed@VM:~/.../padding$ ls
1000.txt      cipher_cbc_f1.bin  cipher_cfb.bin  f1_nopad.txt  f3_nopad.txt
cipher_1000.bin cipher_cbc_f2.bin  cipher_ecb.bin  f1.txt       f3.txt
cipher_cbc.bin  cipher_cbc_f3.bin  cipher_ofb.bin  f2_nopad.txt
[09/18/22]seed@VM:~/.../padding$ bless cipher_1000.bin
Unexpected end of file has occurred. The following elements are not closed: pref
, preferences. Line 22, position 36.
Directory '/home/seed/.config/bless/plugins' not found.
Directory '/home/seed/.config/bless/plugins' not found.
Directory '/home/seed/.config/bless/plugins' not found.
Could not find file "/home/seed/.config/bless/export_patterns".
Could not find file "/home/seed/.config/bless/history.xml".
Document does not have a root element.
Sharing violation on path /home/seed/.config/bless/preferences.xml
Sharing violation on path /home/seed/.config/bless/preferences.xml
Sharing violation on path /home/seed/.config/bless/preferences.xml
```

Changing the 55th bit:

The screenshot shows the Bless debugger interface. The top bar displays the file path: /home/seed/lab2/padding/cipher_1000.bin * - Bless. Below the menu bar is a toolbar with icons for file operations like Open, Save, and Cut/Paste. The main window has two panes. The left pane shows a memory dump of the file cipher_1000.bin. The right pane displays the CPU registers. At the bottom, there are conversion tools for signed/unsigned integers, floating-point numbers, and binary/octal/decimal representations, along with checkboxes for little-endian decoding and ASCII text display.

| | | |
|----------|---|---------------------|
| 00000000 | 01 CA 34 0D FF C2 6E DE 20 43 C5 7E 93 36 FD 03 1A 9B | .4...n. C.~.6.... |
| 00000012 | 4A A4 B0 98 70 39 91 A5 13 09 7E BF F5 6B D9 E4 26 75 | J...p9....~...k.&u |
| 00000024 | 3C A2 91 13 8E 0F 76 77 58 9A 6A 8C 8E 9E 5B 47 EC 71 | <.....vwX.j...[G.q |
| 00000036 | 6E EA 36 A2 AD 36 76 AC F4 F2 C6 18 34 DC C9 43 B7 B7 | n.6..6v.....4..C.. |
| 00000048 | 45 D0 9C EF C8 68 2E 07 01 CA 34 0D FF C2 6E DE 20 43 | E....h....4...n. C |
| 0000005a | C5 7E 93 36 FD 03 1A 9B 4A A4 B0 98 70 39 91 A5 13 09 | ..~.6....J...p9.... |
| 0000006c | 7E BF F5 6B D9 E4 26 75 3C A2 91 13 8E 0F 76 77 58 9A | ~...k.&u<.....vwX. |
| 0000007e | 6A 8C 8E 9E 5B 47 EC 71 6E E3 36 A2 AD 36 76 AC F4 F2 | j...[G.qn.6..6v... |

Signed 8 bit: -22 Signed 32 bit: -365518163 Hexadecimal: EA 36 A2 AD

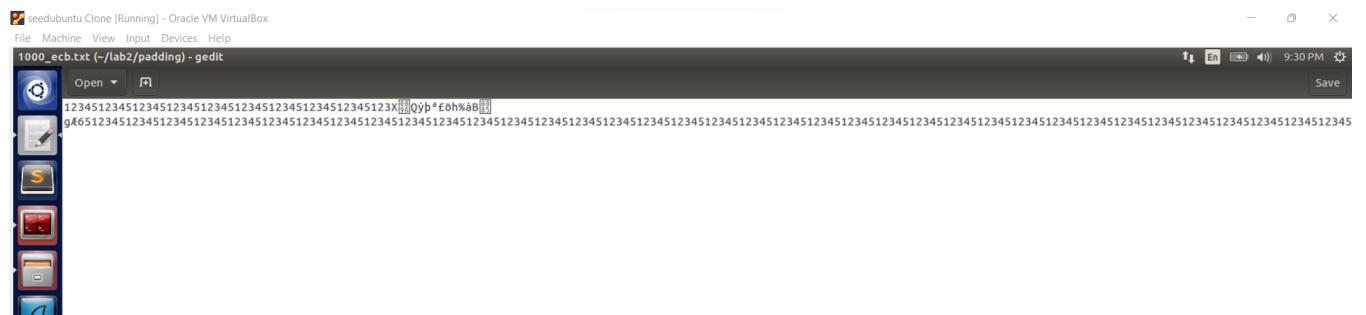
Unsigned 8 bit: 234 Unsigned 32 bit: 3929449133 Decimal: 234 054 162 173

Signed 16 bit: -5578 Float 32 bit: -5.519818E+25 Octal: 352 066 242 255

Unsigned 16 bit: 59958 Float 64 bit: -4.43553821041316E+203 Binary: 11101010 00110110 10

Show little endian decoding Show unsigned as hexadecimal ASCII Text: ?6??

The plaintext file after decrypting the corrupted file:



Here my 1000 bytes file had contents 123451234512345....12345 format so 1 bit change in the ciphertext has changed a part of info in decrypted plain text file.

CBC mode;

Created 1000.txt (1000 bytes) file and then encrypted it as shown below:

```
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cbc -e -in 1000.txt -out cipher_cbc_1000.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
```

Changing the 55th bit:

The file '/home/seed/lab2/padding/cipher_cbc_1000.bin' - Bless

Signed 8 bit: -82 Unsigned 8 bit: 174 Signed 16 bit: -20986 Unsigned 16 bit: 44550

Signed 32 bit: -1375313304 Unsigned 32 bit: 2919653992 Float 32 bit: -3.055547E-11 Float 64 bit: -5.626221916494E-87

Hexadecimal: AE 06 62 68 Decimal: 174 006 098 104 Octal: 256 006 142 150 Binary: 10101110 00000110 01

Show little endian decoding Show unsigned as hexadecimal ASCII Text: ?bh

The file '/home/seed/lab2/pa... Offset: 0x37 / 0x3ef Selection: None OVR

Plain text after decrypting corrupted file:



The whole block of bits is not decrypted in the plain text as in ECB.

CFB:

Created 1000.txt (1000 bytes) file and then encrypted it as shown below:

```
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-cfb -e -in 1000.txt -out cipher_cfb_1000.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
```

Changing the 55th bit:

The screenshot shows the Bless debugger interface with the following details:

- Title Bar:** /home/seed/lab2/padding/cipher_cfb_1000.bin * - Bless
- Toolbar:** Includes icons for file operations (New, Open, Save, Print, Cut, Copy, Paste, Find, Replace, Undo, Redo), zoom, and search.
- Memory Dump:** A table showing memory starting at offset 0x00000000. The first few rows are:

| | | |
|----------|---|----------------------|
| 00000000 | D4 51 DA AA B4 15 75 1F 40 36 10 B9 74 5A 76 B6 9D 39 | .Q....u.@6..tzv..9 |
| 00000012 | DC 40 68 5D 82 93 34 09 4C 58 94 2D 6C 17 0C 28 76 E8 | .@h]..4.LX.-l..(v.. |
| 00000024 | 01 B1 D8 8D 1C B4 B0 32 BE 06 2B 14 05 84 C9 67 4A 20 |2..+....gJ |
| 00000036 | F3 CA 82 EF C8 C0 EC AC BC EF D8 21 BC 28 19 8C 13 3F |!....? |
| 00000048 | A0 EB 9A C5 A9 B1 57 AE C7 5D 30 9F 50 3D 6A 6D DD E3 |W..]0.P=jm.. |
| 0000005a | 0C 36 BD A7 21 72 26 26 12 74 00 AD 41 78 EA E2 8D C8 | .6...!r&&.t..Ax.... |
| 0000006c | A3 9D B4 39 B4 06 DE 5B E0 65 70 D0 10 CA E9 52 57 0F | ...9...[.ep....RW. |
| 0000007e | F2 AB EA 35 88 07 4F 34 6C BD 03 10 28 E7 DA 01 2E 66 | ...5..041...(.f |
- Conversion Controls:** Buttons for Signed 8 bit (-54), Unsigned 8 bit (202), Signed 16 bit (-13694), Unsigned 16 bit (51842), Decimal (202 130 239 200), Octal (312 202 357 310), Binary (11001010 10000010 11), Hexadecimal (CA 82 EF C8), Float 32 bit (-4290532), Float 64 bit (-8.85630567082176E+50), ASCII Text (????), and checkboxes for Show little endian decoding and Show unsigned as hexadecimal.
- Status Bar:** Shows Offset: 0x37 / 0x3e7, Selection: None, and OVR.

The plaintext file after decrypting the corrupted file:



OFB mode:

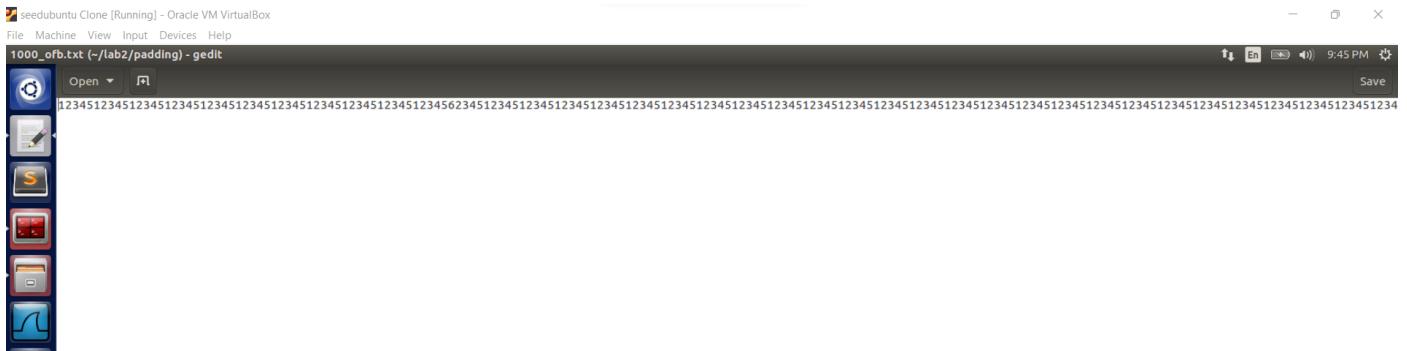
Created 1000.txt (1000 bytes) file and then encrypted it as shown below:

```
[09/18/22]seed@VM:~/.../padding$ openssl enc -aes-128-ofb -e -in 1000.txt -out cipher_ofb_1000.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809  
[09/18/22]seed@VM:~/.../padding$
```

Changing the 55th bit:

The screenshot shows the Bless hex editor interface. The file 'cipher_ofb_1000.bin' is open. The hex dump area shows a series of bytes, with the byte at offset 0x3e7 highlighted in red as '9A'. To the right of the hex dump, the corresponding ASCII representation '.....u.@6..tZv...' is shown. Below the hex dump, there are several input fields for different data types: Signed 8 bit (-102), Unsigned 8 bit (154), Signed 16 bit (-26072), Unsigned 16 bit (39464), and so on. There are also checkboxes for 'Show little endian decoding' and 'Show unsigned as hexadecimal'. At the bottom, there are buttons for 'Offset: 0x37 / 0x3e7', 'Selection: None', and 'OVR'.

Only the 55th bit is affected in the decrypted file:



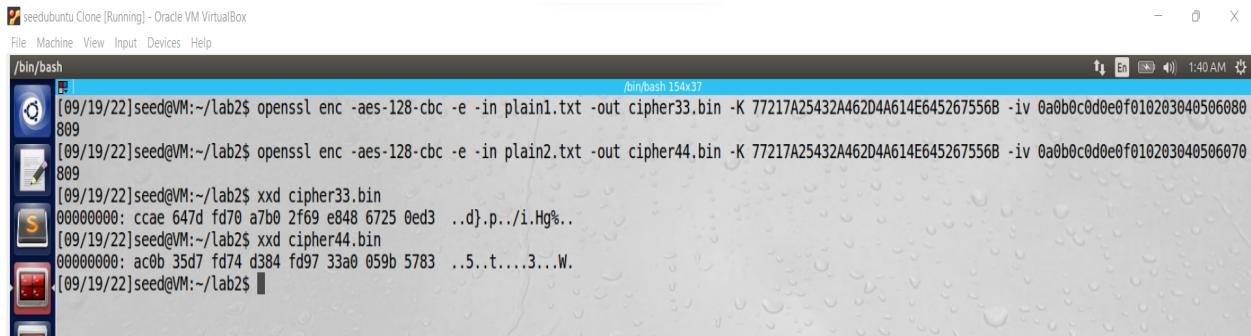
2.6.1). Task 6: Initial Vector(IV)

6.1) I first created two plaintexts plain1.txt and plain2.txt each of them contain 5 bytes(12345) I encrypted them using same key(77217A25432A462D4A614E645267556B) and initial vector(0a0b0c0d0e0f010203040506070809) as shown below with the contents of the ciphertext:

The screenshot shows a terminal window with the following session:

```
[09/19/22]seed@VM:~/lab2$ openssl enc -aes-128-cbc -e -in plain1.txt -out cipher11.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/19/22]seed@VM:~/lab2$ openssl enc -aes-128-cbc -e -in plain2.txt -out cipher22.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070809
[09/19/22]seed@VM:~/lab2$ xxd cipher11.bin
00000000: ac0b 35d7 fd74 d384 fd97 33a0 059b 5783 ..5..t....3...W.
[09/19/22]seed@VM:~/lab2$ xxd cipher22.bin
00000000: ac0b 35d7 fd74 d384 fd97 33a0 059b 5783 ..5..t....3...W.
[09/19/22]seed@VM:~/lab2$
```

Then I encrypted same plain text files using different Initial Vectors:



```
seedubuntu Clone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
/bin/bash
[09/19/22]seed@VM:~/lab2$ openssl enc -aes-128-cbc -e -in plain1.txt -out cipher33.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506080
809
[09/19/22]seed@VM:~/lab2$ openssl enc -aes-128-cbc -e -in plain2.txt -out cipher44.bin -K 77217A25432A462D4A614E645267556B -iv 0a0b0c0d0e0f010203040506070
809
[09/19/22]seed@VM:~/lab2$ xxd cipher33.bin
00000000: ccae 647d fd70 a7b0 2f69 e848 6725 0ed3 ..d}.p../.Hg%..
[09/19/22]seed@VM:~/lab2$ xxd cipher44.bin
00000000: ac0b 35d7 fd74 d384 fd97 33a0 059b 5783 ..5.t....3...W.
[09/19/22]seed@VM:~/lab2$
```

The encrypted files were not the same and were completely different from each other.

2.6.2) Common Mistakes: Use the same IV

In the Boolean algebra we know that if $A \oplus B = C$ then $A \oplus C = B$.

Let's use the above rule to find the plaintext P2.

Given Plaintext P1 : This is a known message!

Plaintext P1 converted to Hex: 546869732069732061206b6e6f776e206d6573736167652100

Then plaintext is padded with 2 zeros as it is 2bytes smaller than cipher text.

Ciphertext C1 : a469b1c502c1cab966965e50425438e1bb1b5f9037a4c15913

Now if we XOR P1 and C1 we get the output of the block cipher(Which takes key and IV as input) and let's call this output as Ks

Ks : f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a47813

The catch here is this is given as input to the next block and since its mentioned IVs are the same then the output of the second block cipher will also lead to the same Ks. So if we XOR Ks and Ciphertext we will be able to retrieve Plain text P2.

P2= Ks XOR C2= 4f726465723a204c61756e63682061206d697373696c652100

Hex to text of P2= Order: Launch a missile!

Thus, I was able to recover full P2.

In the same question if CFB is there then also we can recover full Plaintext.

2.6.3) Task 6.3. Common Mistake: Use a Predictable IV

The encryption method used is 128-bit AES with CBC mode.

Key (in hex): 00112233445566778899 aabbccddeeff (known only to Bob)

Ciphertext (C1): bef65565572ccee2a9f9553154ed9498 (known to both) | V used on P1
(known to both)(in ascii): 1234567890123456 (in hex) :
31323334353637383930313233343536

Next IV (known to both)(in ascii): 1234567890123457 (in hex) :
31323334353637383930313233343537

P2(hex)= 31323334353637383930313233343536 XOR
31323334353637383930313233343537 XOR 596572

P2(hex):596572

Since it is known that plane text is either yes or no, we can find the hex similar to the p2 to conclude it to be the required value in this case P2 is “yes”

In this case 1 = yes and 0 = no,

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: hexadecimal (base 16) ↴

31323334353637383930313233343536

II. Input: hexadecimal (base 16) ↴

31323334353637383930313233343537

Calculate XOR

III. Output: hexadecimal (base 16) ↴

1

2.8 Task 7: Programming using the Crypto Library - Extra Credit 5%

Code:

```
#include <openssl/conf.h>  
  
#include <openssl/evp.h>  
  
#include <openssl/err.h>  
  
#include <string.h>
```

```
void handleErrors(void);

void pad(char *s, int length);

int strcicmp(char const *a, char *b);

int main (void)

{

    unsigned char match[] = "MATCH";

    unsigned char nomatch[] = "NOMATCH";

    int i,outlen,tmplen;

    unsigned char outbuf[1024 + EVP_MAX_BLOCK_LENGTH];

    FILE *key;

    EVP_CIPHER_CTX ctx;

    EVP_CIPHER_CTX_init(&ctx);

    /*

     * Set up the key and iv. Do I need to say to not hard code these in a

     * real application? :-)

     */

    /* A 128 bit IV */
```

```
unsigned char iv[16] ="010203040506070809000a0b0c0d0e0f";  
  
/* Message to be encrypted */  
  
char inText[]="This is a secret tool";  
  
  
key= fopen("words.txt", "r");  
  
  
/*  
 * Buffer for ciphertext. Ensure the buffer is long enough for the  
 * ciphertext which may be longer than the plaintext, depending on the  
 * algorithm and mode.  
 */  
  
char ciphertext[]=  
"ece6753e938f8f903cabbbe12d395bf5f7eaе38ad918a2d3e1c3a832476d5c7a";  
  
  
  
  
while (fgets(words,16, key))  
{  
    i= strlen(words);  
  
    words[i-1]='\0';  
  
    i= strlen(words);  
  
    if(i<16)  
    {  
        pad(words, (16));  
    }  
}
```

```
    EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, words, iv);

    if(!EVP_EncryptUpdate(&ctx, outbuf, &outlen, InText, strlen(intext)))
    {
        EVP_CIPHER_CTX_cleanup(&ctx);
        return 0;
    }

    if(!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmpplen))
    {
        EVP_CIPHER_CTX_cleanup(&ctx);
        return 0;
    }

    outlen += tmpplen;
```

```
int j;

char *buf_str= (char*) malloc (2*outlen +1);

char *buf_ptr= buf_str;

for(i=0; i<outlen;i++) {

    buf_ptr += sprintf(buf_ptr, "%02X", outbuf[i]);

}

*(buf_ptr +1)= '\0';

if(strcicmp(ciphertext,buf_str)==0) {

    printf("%s\n", words)

}

fclose(key);

return 1;

}

void handleErrors(void)

{

    ERR_print_errors_fp(stderr);

    abort();

}
```

```
void pad(char *s, int length)

{

    int l;

    l=(strlen(s));



    while(l<length) {

        s[l] = ' ';

        l++;

    }

    s[l]= '\0';

}

int strcicmp(char const *a, char const *b){

    for(;;a++;b++) {

        int d= tolower(*a)- tolower(*b);

        if (d!=0 || !*a) {

            return d;

        }

    }

}
```

