AI331: Information Security & Cryptography
Lab 7 — Implementing the Affine Cipher

Instructor: Yugarshi Shashwat

Learning Objectives

By the end of this lab, you will be able to:

- Implement the Affine substitution cipher over the alphabet *{A, . . . , Z}* with correct key validation.
- Compute modular inverses and use them to decrypt affine ciphertexts. • Perform a basic

cryptanalysis of Affine ciphers using two-letter mapping or frequency heuristics.

Pre-reads

Stinson, *Cryptography: Theory and Practice*, Ch. 1–2 (classical ciphers), course notes on modular arithmetic.

1 Affine Cipher: Definition

Let letters map to integers via *A* 7→ 0*, B* 7→ 1, . . . , *Z* 7→ 25. For a key (a, b) with $a \in \{1, . . . , 25\}$, $b \in \{0, . . . , 25\}$ and gcd(a, 26) = 1, define:

$$E_{a,b}(x) \equiv (ax + b) \bmod 26, \quad D_{a,b}(y) \equiv a^{-1}(y - b) \bmod 26,$$

where $a^{-1}$is the multiplicative inverse of a modulo 26.

Valid keys. Since 26 = 2 · 13, the invertible a's are

$$\{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\}.$$

2 I/O & Text Handling Conventions

- Accept plaintext/ciphertext as ASCII strings; ignore characters not in A..Z (or optionally pass them through unchanged).
- Convert letters to uppercase before processing.
- Spaces, punctuation, digits may be removed or preserved (choose and document your choice); tests will follow your documented convention.

3 Task A — Clean Implementation (60 pts)

Goal: Implement encryption and decryption for the Affine cipher with robust key handling. 1

Requirements

1. **Key validation:** Check gcd(a, 26) = 1 at runtime; reject invalid a with a clear error message.
2. **Modular inverse:** Implement extended Euclidean algorithm to compute a^{-1} mod 26. 3.

Encrypt/Decrypt: Functions $E_{a,b}$ and $D_{a,b}$ operating on sanitized text. 4. **CLI interface:**

```
affine --mode enc --a <int> --b <int> --in "<text>"
affine --mode dec --a <int> --b <int> --in "<text>"
```

5. **Tests (include in README or comments):**
   - With (a, b) = (5, 8), HELLOWORLD → RCLLAOAPLX.
   - Decrypting the above must return HELLOWORLD.

Listing 1: AI331 – Lab 7: Affine Cipher (C++ starter skeleton)

```
/*
AI331 Lab 7: Affine Cipher ( C ++ starter )
------------------------------------------
Build : g ++ - O2 - std = c ++17 affine . cpp -o affine
Usage : ./ affine enc a b
./ affine dec a b
Input : Reads a single line of text from STDIN .
Policy : Keeps only letters A .. Z ( uppercased ) . Non - letters are dropped .
*/

# include < bits / stdc ++. h >
using namespace std ;

// ----- small helpers -----
static inline int mod (int x , int m ) {
int r = x % m ;
return ( r < 0) ? r + m : r ;
}
static inline bool is_valid_a ( int a ) {
// Valid a iff gcd (a , 26) == 1
int x = abs ( a ) , y = 26;
while ( y ) { int t = x % y ; x = y ; y = t ; }
return x == 1;
}

// Convert to uppercase A .. Z only , drop everything else
static string sanitize_letters_only_upper ( const string & s ) {
string t ; t . reserve ( s . size () ) ;
for ( char c : s ) if ( isalpha (( unsigned char ) c ) ) t . push_back (( char ) toupper ( c ) ) ;
return t ;
}

// ===== \ todo : modular inverse a ^{ -1} ( mod 26) via extended Euclid =====
// Return the multiplicative inverse of a modulo m (=26 here ) .
```



```
// If gcd (a , m ) != 1 , throw .
static int modinv ( int a , int m ) {
// \ todo : implement extended Euclidean algorithm to find x with a * x = 1 ( mod m )
// Hints :
// - Keep track of ( old_r , r ) , ( old_s , s ) pairs
// - When loop ends , old_r = gcd (a , m ) and old_s is the inverse mod m
// - Return mod ( old_s , m )
```

```
throw runtime_error ( " modinv : TODO " ) ;
}

// ===== \ todo : encrypt =====
// For each letter X in 0..25 , compute Y = ( a * X + b ) mod 26 and map back to 'A '+ Y .
static string encrypt_text ( const string & plain , int a , int b ) {
// \ todo : implement E_ {a , b }( x ) = ( a * x + b ) mod 26 over sanitized uppercase text
// Steps :
// - For each character c in ' plain ', map x = c - 'A '
// - y = ( a * x + b ) mod 26
// - push_back ( ' A ' + y )
// Return the resulting ciphertext .
return " TODO_ENCRYPT " ;
}

// ===== \ todo : decrypt =====
// Use a_inv = a ^{ -1} ( mod 26) : X = a_inv * ( Y - b ) mod 26.
static string decrypt_text ( const string & cipher , int a , int b ) {
// \ todo : compute a_inv using modinv (a , 26)
// \ todo : for each cipher letter y = c - 'A ', compute x = a_inv * ( y - b ) mod 26
// \ todo : map back to 'A ' + x
return " TODO_DECRYPT " ;
}

int main (int argc , char ** argv ) {
ios :: sync_with_stdio ( false ) ;
cin . tie ( nullptr ) ;

if ( argc != 4) {
cerr << " Usage : " << argv [0] << " < enc | dec > a b \ n " ;
return 1;
}

string mode = argv [1];
int a = stoi ( argv [2]) ;
int b = stoi ( argv [3]) ;

if (! is_valid_a ( a ) ) {
cerr << " Error : a must be coprime with 26. Choose a in {1 ,3 ,5 ,7 ,9 ,11 ,15 ,17 ,19 ,21
,23 ,25}.\ n " ;
return 1;
}
b = mod (b , 26) ;

string line ;
if (! getline ( cin , line ) ) line = " " ;
```



```

// Sanitize : keep only A .. Z ( uppercased )
string in = sanitize_letters_only_upper ( line ) ;

try {
if ( mode == " enc " ) {
cout << encrypt_text ( in , a , b ) << " \ n " ;
} else if ( mode == " dec " ) {
cout << decrypt_text ( in , a , b ) << " \ n " ;
} else {
cerr << " Mode must be ' enc ' or ' dec '.\ n " ;
return 1;
}
} catch ( const exception & e ) {
cerr << " Error : " << e . what () << " \ n " ;
return 1;
}
```

```
return 0;
}

/*
---------------------------
Minimal self - check ( manual )
---------------------------
Example ( once TODOs are done ) :

$ echo HELLOWORLD | ./ affine enc 5 8
RCLLAOAPLX

$ echo RCLLAOAPLX | ./ affine dec 5 8
HELLOWORLD
*/
```

4 Task B — Cryptanalysis & Key Recovery (40 pts) Goal:

Recover (a, b) and decrypt given ciphertext(s) produced by an Affine cipher.

B1. Two-letter mapping (deterministic)

If you can guess two plaintext–ciphertext letter correspondences $(x_1 \to y_1)$ and $(x_2 \to y_2)$ with $x_1 \neq x_2$, solve

$$y_1 \equiv ax_1 + b \pmod{26},$$
$$y_2 \equiv ax_2 + b \pmod{26}.$$

Subtract to eliminate b: $(y_1 - y_2) \equiv a(x_1 - x_2) \pmod{26}$. If $\gcd(x_1 - x_2, 26) = 1$, then $a \equiv$

$$(y_1 - y_2) \cdot (x_1 - x_2)^{-1} \pmod{26},\ b \equiv y_1 - ax_1 \pmod{26}.$$

B2. Frequency heuristic (practical)

Assume E and T are the most common plaintext letters in English. Let y_{max} and y_{2nd} be the two most frequent cipher letters. Try the two pairings:

$$(E \mapsto y_{max},\ T \mapsto y_{2nd}) \text{ and } (E \mapsto y_{2nd},\ T \mapsto y_{max})$$

Solve for (a, b) via B1 for each pairing; score the resulting decryptions with a dictionary or bigram frequencies, and choose the better one.

Deliverables for Task B.

1. A small script/function that, given ciphertext, attempts the frequency method above and prints candidate keys & plaintext.

2. A short writeup (5–10 lines) describing which strategy succeeded and why.

5 Test Vectors

Key (a, b)	Plaintext	Ciphertext
(5, 8)	HELLOWORLD	RCLLAOAPLX

(7, 3) CRYPTOGRAPHY VEUFWBRDUCBK

6 Edge Cases & Notes

• If you choose to *preserve* non-letters, ensure decryption restores them in place. • Reject invalid a immediately; do not attempt decryption without a^{-1}. • Document your sanitization policy at the top of your source file(s).

7 What To Submit

1. README.md: build/run instructions; text-handling policy; brief summary of Task B approach. 2. Source files: affine.cpp
3. Optional: small test script and sample outputs.

Academic Integrity: Write your own code. Cite any external references used.

8 Background: Extended Euclidean Algorithm

To decrypt in the Affine cipher, we need the multiplicative inverse a^{-1} mod 26. This requires solving the congruence

$$a \cdot a^{-1} \equiv 1 \pmod{26}.$$

The method to compute such inverses is the **Extended Euclidean Algorithm** (EEA).

The Euclidean Algorithm

Recall that for integers a, b with $a > b > 0$, the Euclidean algorithm finds $\gcd(a, b)$ by repeated division:

$$a = q_1 b + r_1,\ b = q_2 r_1 + r_2,\ r_1 = q_3 r_2 + r_3,\ \ldots$$

until some remainder is 0. The last nonzero remainder is $\gcd(a, b)$.

Extended version

The extended form not only computes $\gcd(a, b)$, but also integers x, y such

that $ax + by = \gcd(a, b)$.

This is called a *Bézout identity*. If $\gcd(a, b) = 1$, then x is precisely the inverse of $a \pmod{b}$.

Algorithm sketch

1. Initialize $(\text{old\_r}, r) = (a, b)$, $(\text{old\_s}, s) = (1, 0)$, $(\text{old\_t}, t) = (0, 1)$.

2. While $r \neq 0$:

$$q = \lfloor \text{old\_r}/r \rfloor,\ (\text{old\_r}, r) \leftarrow (r, \text{old\_r} - q \cdot r),$$

and update s, t similarly:

$$(\text{old\_s}, s) \leftarrow (s, \text{old\_s} - q \cdot s), (\text{old\_t}, t) \leftarrow (t, \text{old\_t} - q \cdot t).$$

3. When the loop ends, old_r = gcd(a, b), and (old_s, old_t) satisfy $a \cdot \text{old\_s} + b \cdot \text{old\_t} = \gcd(a, b)$.

Worked example

Find a^{-1} for $a = 5 \pmod{26}$.

$$26 = 5 \cdot 5 + 1.$$

So $1 = 26 - 5 \cdot 5$. Rearrange: $1 = (-5) \cdot 5 + 1 \cdot 26$.
Thus $x = -5$ is a solution: $5 \cdot (-5) \equiv 1 \pmod{26}$. Taking modulo 26, we get $a^{-1} = 21$.

Check: $5 \cdot 21 = 105 \equiv 1 \pmod{26}$.

Hence, the extended Euclidean algorithm lets us compute the inverse a^{-1} efficiently for any valid a in the Affine cipher.